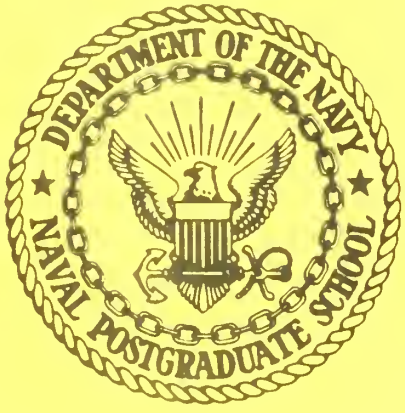
NPS52-83-003

NAVAL POSTGRADUATE SCHOOL

Monterey, California



THE IMPLEMENTATION OF A MULTI-BACKEND DATABASE SYSTEM (MDBS): PART III - THE MESSAGE-ORIENTED VERSION WITH CONCURRENCY CONTROL AND SECONDARY-MEMORY-BASED DIRECTORY MANAGEMENT

Richard D. Boyne, Steven A. Demurjian, David K. Hsiao, Douglas S. Kerr and Ali Orooji

March 1983

epared for: Chief of Naval Research
Arlington, VA 22217

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral J. J. Ekelund
Superintendent

David A. Schrady
Acting Provost

The work report herein was supported by Contract N00014-75-C-0573 from The Office of Naval Research.

This report was prepared by:

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: NPS52-83-003	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE *(and Subtitle)*: THE IMPLEMENTATION OF A MULTI-BACKEND DATABASE SYSTEM (MDBS): PART III - THE MESSAGE-ORIENTED VERSION WITH CONCURRENCY CONTROL AND SECONDARY-MEMORY-BASED DIRECTORY MANAGEMENT	
5. TYPE OF REPORT & PERIOD COVERED: Technical Report	
6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s): Richard D. Boyne, Steven A. Demurjian, David K. Hsiao, Douglas S. Kerr and Ali Orooji	
8. CONTRACT OR GRANT NUMBER(s)	
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School, Monterey, CA 93940	
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 4115-A1	
11. CONTROLLING OFFICE NAME AND ADDRESS: Chief of Naval Research, Arlington, VA 22217	
12. REPORT DATE: March 1983	
13. NUMBER OF PAGES: 89	
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*	
15. SECURITY CLASS. *(of this report)*: Unclassified	
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE	

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

backend database system, database system implementation, database computer, database machine, software engineering, database

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

The multi-backend database system (MDBS) uses one minicomputer as the master or controller, and a varying number of minicomputers and their disks as slaves or backends. MDBS is primarily designed to provide for database growth and performance enhancement by the addition of identical backends. No special hardware is required. The backends are configured in a parallel manner. A new backend may be added by replicating the existing software on the new backend. No new programming or reprogramming is required.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

A prototype MDBS is being implemented in order to carry out design verification and performance evaluation. This report is the third in a series which describes the MDBS implementation. In the report, an overview of MDBS and its process structure is first given. Then, the inter-process communication (messages between processes within a minicomputers, i.e., the controller and a backend or any two backends) are described.

The processes in the MDBS controller (request preparation, insert information generation and post processing) and two of the three processes in the MDBS backends (directory management and record processing) were described in the previous reports. The third process in the MDBS backends, namely, concurrency control, is described in detail in this report.

We have implemented two versions of the directory-management process. The primary-memory-based directory management was implemented earlier and described in the previous reports. The secondary-memory-based directory management is described in this report.

Software engineering techniques have been employed throughout the MDBS development. The techniques and their effectiveness were described in the previous reports. The problems that we have observed during the development, integration and testing of MDBS are described in this report.

The appendices contain the detailed design for concurrency control.

TABLE OF CONTENTS

PREFACE v
LIST OF FIGURES vi
1. INTRODUCTION 1
1.1 A Revised Implementation Strategy - What and Why? 1
1.1.1 The Original Strategy 4
1.1.2 The Revised Versions 4
1.2 The Software Engineering Experience 6
2. MESSAGE PASSING IN THE MULTI-BACKEND DATABASE SYSTEMS (MDBS) 7
2.1 Message Passing Within a Backend 7
2.2 Message Passing Within the Controller 10
2.3 Message Passing Between Computers 10
2.4 MDBS Message Descriptions 14
2.4.1 MDBS Message Definitions 17
2.4.2 Request Execution in MDBS - Viewed via Message Passing 26
(A) Sequence of Actions for an Insert Request 28
(B) Sequence of Actions for a Delete Request 28
(C) Sequence of Actions for a Retrieve Request with Aggregate Operator 28
(D) Sequence of Actions for an Update Request Causing an Updated Record to Change Cluster 31
3. CONCURRENCY CONTROL 35
3.1 Two Types of Consistency 35
3.2 Three Categories of Locks 37
3.3 The Notion of Transaction 38
3.4 Concurrency Control Using a Message-Oriented Approach 39
3.4.1 The Process Structure in the Backends 39
3.4.2 Cluster-To-Traffic-Unit Table (CTUT) 40
3.4.3 Traffic-Unit-To-Cluster Table (TUCT) 40
3.4.4 The Processing of Concurrency Control Information 42
(A) The Processing of a New Traffic Unit 42
(B) The Processing of a Finished Request 44
(C) The Conversion of Locks 44
(D) Future Modifications 46
4. THE SECONDARY-MEMORY-BASED DIRECTORY MANAGEMENT 48
4.1 The Attribute Table (AT) 48
4.2 The Descriptor-to-Descriptor-Id Table (DDIT) 52

4.3 The Cluster-Definition Table (CDT) 55
4.3.1 The Descriptor-Id-and-Cluster-Id-Bit-Map Table (DCBMT) 59
4.3.2 The Cluster-Id-To-Secondary-Storage-Address Table (CSSAT) .. 59
5. OUR IMPLEMENTATION EXPERIENCE 65
5.1 System Issues 65
5.1.1 The Controller 65
5.1.2 The Backends 65
5.1.3 The Test Interface 67
5.1.4 The Disk Input/Output Interface 67
5.2 Our Software-Engineering Observations 68
5.2.1 Use of Standards in Coding 68
5.2.2 The System Programming Language and the Language Compiler .. 69
5.2.3 High-Level Design vs. Low-Level Design 72
5.2.4 The Importance of Interfaces Between Processes 72
5.2.5 The Problems in a University Environment 73
REFERENCES 74
APPENDIX A - HOW TO READ AND FOLLOW THE PROGRAM SPECIFICATIONS 75
A.1 Parts within an Appendix 75
A.2 The Format of a Part 75
A.3 Documentation Techniques for a Part 76
APPENDIX B : THE SSL SPECIFICATIONS FOR MDBS CONCURRENCY CONTROL 77

PREFACE

This work was supported by Contract N00014-75-C-0573 from the Office of Naval Research to Dr. David K. Hsiao and conducted in the Laboratory for Database Systems Research. The Laboratory for Database Systems Research is initially funded by the Digital Equipment Corporation (DEC), Office of Naval Research (ONR) and the Ohio State University (OSU) and consists of the staff, graduate students, undergraduate students, visiting scholars and faculty for conducting research in database systems. Dr. Douglas S. Kerr, Associate Professor of Computer and Information Science at the Ohio State University, is the present Director of the Laboratory.

Since July 1, 1982, Dr. Hsiao assumed the Chairmanship of the Computer Science Department at the Naval Postgraduate School and continued the funded research at the Naval Postgraduate School. The Laboratory for Database Systems Research will be moved to the Naval Postgraduate School (NPS) in June of 1983 and supported by DEC, ONR, and NPS. This technical report was drafted at the Ohio State University and completed at the Naval Postgraduate School.

We would like to thank all those who have helped with the MDBS project. In particular, the MDBS design and analysis were developed by Jai Menon. (Now, Dr. Jai Menon of IBM Research Laboratory, San Jose, California.) He also provided much help in the detailed designs. A visiting scholar, Xing-Gui He, is involved with MDBS project. Several undergraduate students are also involved with the project: Raymond Browder, Chris Jeschke, Jim McKenna, and Joe Stuber. Several graduate students, visiting scholars and undergraduate students provided much help in the detailed design and coding: Steven Barth, Julie Bendig, Abdulrahim Beram, Patti Dock, Masanobu Higashida, Jim Kiper, Drew Logan, William Mielke, Tamer Ozsu, Zong-Zhi Shi, and Paula Strawser. Jose Alegria, Tom Bodnovich and David Brown contributed background material which was necessary for making our design decisions. We would also like to thank the laboratory staff and other operators who provided us with system support: Bill Donovan, Doug Karl, Paul Placeway, Steve Romig, Jim Skon, Dennis Slaggy, Mark Verber, and Geoff Wyant.

LIST OF FIGURES

Figure 1 - The MDBS Hardware Organization 2
Figure 2 - The MDBS Process Structure 3
Figure 3a - Conceptual Flow for Process A Sending a Message to Process B 8
Figure 3b - Backend Message Passing 9
Figure 4a - VAX Inter-process Communication Using Mailboxes 11
Figure 4b - Controller Message Passing 12
Figure 5a - MDBS Message Passing : The Inter-Machine View 13
Figure 5b - Message Passing from Machine A to Machine B 15
Figure 6 - MDBS General Message Format 16
Figure 7 - The MDBS Message Types 18
Figure 8 - MDBS Controller and Backend Configuration 19
Figure 9 - Controller Related Messages 20
Figure 10 - REQP, IIG (Controller); DM (Backend) Related Messages 22
Figure 11 - REQP, RECP and PP Related Messages 23
Figure 12 - (Backend) DM and RECP Related Messages 25
Figure 13 - (Backend) DM, RECP and CC Related Messages 27
Figure 14 - Sequence of Message Passing Events for an Insert Request 29
Figure 15 - Sequence of Message Passing Events for a Delete Request 30
Figure 16 - Sequence of Message Passing Events for a Retrieve Request With an Aggregate Operator 32
Figure 17 - Sequence of Message Passing Events for an Update Request That Causes a Record to Change Cluster 33
Figure 18 - A Sample of Cluster-To-Traffic-Unit Table (CTUT) 41
Figure 19 - The Traffic-Unit-To-Cluster Table (TUCT) Corresponding to the CTUT in Figure 18 43
Figure 20 - The Attribute Table (AT) and its Relationship to the Descriptor-To-Descriptor-Id Table (DDIT) 49
Figure 21 - A Sample Attribute Table Stored as a B-tree 50
Figure 22 - An Example of Lookup in an Attribute Table 51
Figure 23 - A Sample B+tree Containing the Descriptors for a Directory Attribute 53
Figure 24 - An Example of The Descriptor Search Phase 54
Figure 25 - A Sample of The Cluster-Definition Table (CDT) 56
Figure 26 - A Sample of The Descriptor-Id-Cluster-Id-Bit-Map Table (DCBMT) 57
Figure 27 - A Sample of The Cluster-Id-to-Secondary-Storage-Address Table (CSSAT) 58

Figure 28 - A Sample DCBMT Stored as a Linked List 60
Figure 29 - An Example of The Cluster Search Phase 61
Figure 30 - A Sample CSSAT Stored as an Indexed-Linked List 63
Figure 31 - An Example of The Address Generation Phase 64

1. INTRODUCTION

This report is the the third in a series describing the implementation of MDBS, a multi-backend database system [Kerr82, He82]. The original design was given in [Hsia81a, Hsia81b]. It is assumed that the reader is already familiar with these earlier reports. We will, however, give a very brief review of the MDBS design.

An overview of MDBS hardware organization is shown in Figure 1. MDBS is connected to a host computer through the controller. The controller and backends are, in turn, connected by a broadcast bus. The controller receives requests from a host computer. It then broadcasts each request to all backends at the same time over the bus. The data from each aggregate, e.g., a file or relation, is distributed across backends. Thus all backends can execute a request in parallel.

The process structure of MDBS is shown in Figure 2. A major design goal for MDBS was to minimize the work done by the controller and to maximize the work done by the backends. The controller must, however, perform some functions. It must first prepare a request for execution by the backends. This function is performed by request preparation. The controller must also coordinate responses from the backends. This function is performed by post processing. In addition, for consistency reasons, certain functions required for record insertion must also be performed in the controller. These functions are performed by insert information generation.

As much work as possible has been given to the backends, this work consists of three categories of functions: directory management, concurrency control and record processing. The directory management functions are used to determine the addresses of the records required to process a particular request. The concurrency control function allows concurrent access to the database by different requests. The record processing functions perform the actual data retrieval and storage as well as the processing required on any particular record (e.g., the computation of a maximum value).

1.1. A Revised Implementation Strategy - What and Why?

The original implementation strategy called for the implementation of five different versions of MDBS, beginning with a very simple system using a

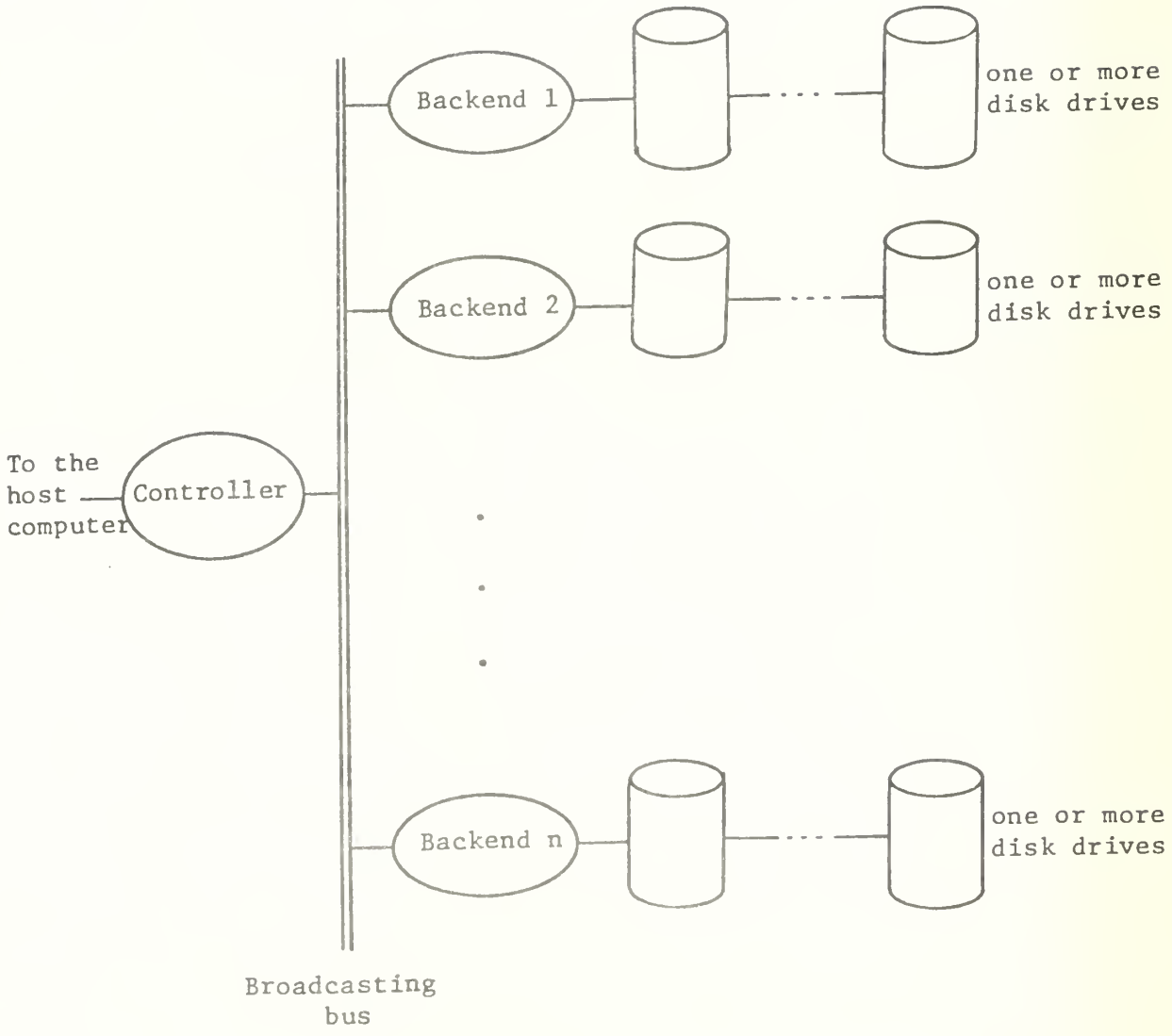


Figure 1. The MDBS Hardware Organization

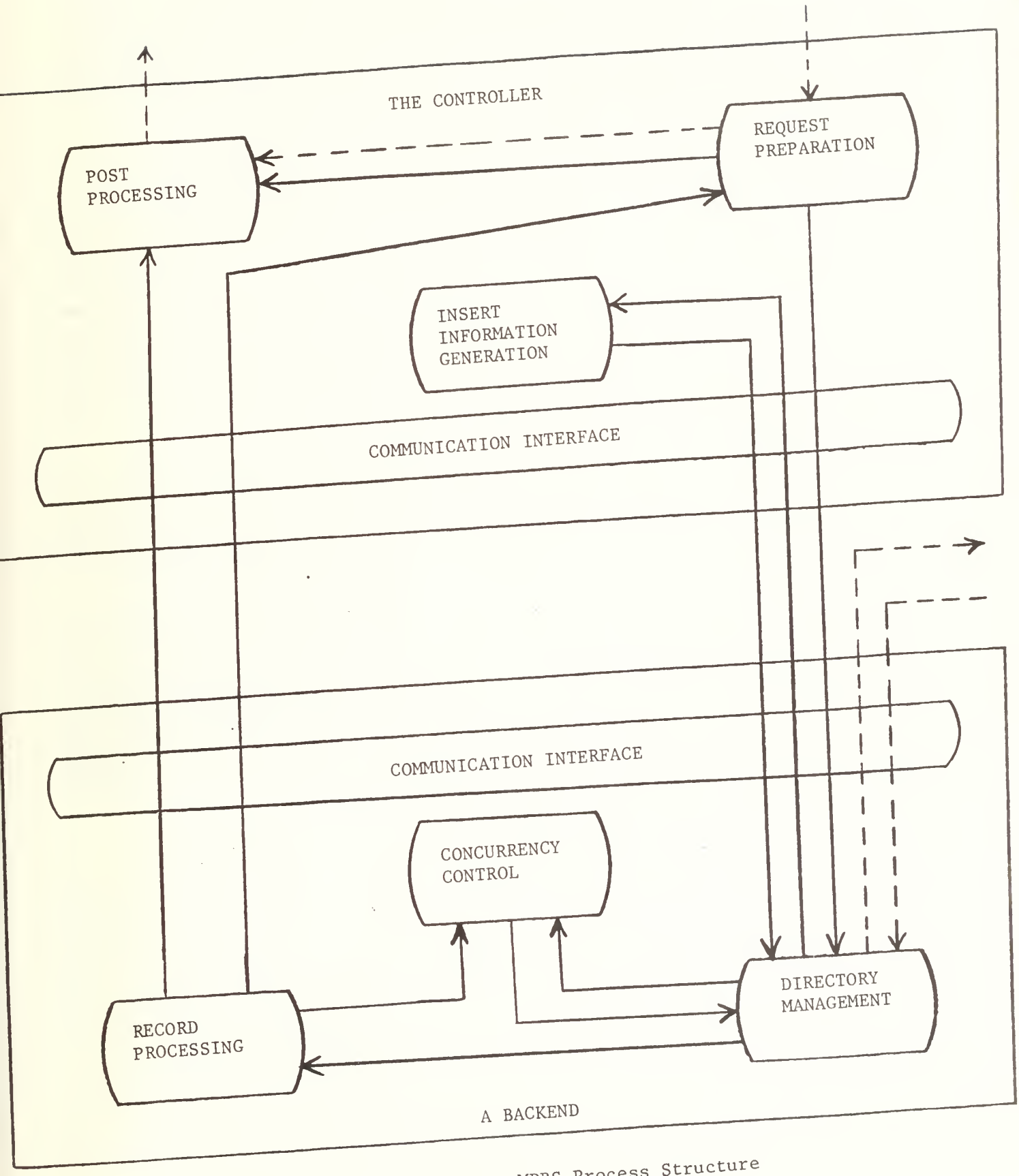


Figure 2. The MDBS Process Structure

single minicomputer without concurrency control and with a simplified directory management and ending with a full system including all the designed features. In this section we will discuss how and why we have revised this strategy somewhat.

1.1.1. The Original Strategy

As stated above, the first system was to be a simple system running as a single process on a single computer. The purpose of this version was to allow the development of much of the functionality of MDBS in an environment that would allow easy development and debugging. It would allow us to delay the implementation of inter-computer communication functions and to learn the systems programming language and environment. Directory management was to be simplified by assuming that all directory information could be stored in the primary memory.

The second version was to include concurrency control, but still be restricted to one minicomputer. The problem of inter-process communication was to be attacked. However we planned to avoid the problems associated with inter-computer communication.

The third version was to run the same functions on multiple computers by adding the inter-computer communication functions. This was to be the first "real" system. The fourth version was to include a "good" directory management system, one using secondary storage for the directory data. The fifth and final version was to include access control in the backends and a friendly user interface in the controller or in a host computer.

1.1.2. The Revised Versions

Because of the experience gained during the implementation, we have chosen to revise our implementation strategy somewhat. We are still implementing version III (The real system with simplified directory management) and version IV(the real system with "good" directory management). However, we have chosen to modify the strategy of how to get to version III. In order to avoid confusion, we will call the new versions A, B, and so on.

Version A was implemented on a VAX-11/780 running UNIX. It included the request preparation and insert information generation functions of the controller. The post-processing functions were not implemented. Instead, the

output was displayed directly from record processing, which is a function of the backend. The directory management and record processing functions of the backends were implemented, except that the aggregate operations, such as average, minimum and maximum were omitted since they are straightforward but not critical to the overall system implementation effort. In addition, the routines that were to perform the actual input and output of data to and from the disks were not implemented. These routines were omitted because they are operating-system dependent and this version was to run on a VAX using UNIX whereas the actual backends were to be PDP-11/44s using RSX-11M. Since the "database" was not to be stored on disks in this version, we implemented a pseudo disk using the main memory. In addition a user test interface was implemented. This version which was implemented as a single process was called version A.

The next step we chose, version B, was to convert to a multi-process system which would have the same functionality as version A. As explained in the previous report [He82], we chose to implement each category of functions as a separate process. Thus we would have three processes in the controller: request preparation, insert information generation and post processing. In this version the backends would have two processes: directory management and record processing. Concurrency control would be added as a third process later. The conversion from a single process to a multi-process implementation required some modifications of the code in version A. In particular, the programs that simulated message passing had to be replaced by programs that sent and received messages between processes.

Because of the available inter- and intra-computer message passing facilities, we decided to modify our plans for the second version of MDBS. In particular, we chose to use two computers, a VAX and a PDP-11/44, rather than one. The final version of MDBS was to use PDP-11/44s (using RSX-11M) for the backends and a VAX (using VMS) for the controller. The inter-process communication facilities of VMS and RSX-11M are different [DEC79b, DEC80]. Thus implementing all the processes on one computer, the VAX say, would then have required us to change the message passing programs later. In addition, the communication software to support the communication between computers had been implemented by this time. Thus we chose to use two computers in version B. The inter-computer communication functions were implemented as two processes, a

sender and a receiver. Their implementation is discussed in Chapter 2.

As just described, version B used only a single backend. Thus we converted to two backends for version C. This version ran on three computers, a VAX and two PDP-11/44s. However it still lacked several required functions. There was no concurrency control. In addition all the data, both the database itself as well as the directory, was stored in primary memories. Thus no disk input and output was required.

By changing from using a simulated disk in version C to an actual disk system, we obtained version D. This change, though logically simple, was difficult to implement since it required us to create a low level interface with the operating system of the PDP-11/44s. This interface is discussed in Chapter 5. Version D included all the functions we had intended for our first real system, version III, except concurrency control. Thus we next added a concurrency control process to give us version E. This process is described in Chapter 3. This version is the same as our original version III.

The next step in our implementation, version F, is to change directory management so that directory information is stored on the secondary memory rather than in the primary memory. This change is complex, since restructuring of the directory data is also required. The secondary-memory-based directory management of version F is described in Chapter 4.

As in the original plan, the final version, now version G, will incorporate access control in the backends and a friendly user-interface in the controller or host computer.

1.2. The Software Engineering Experience

We have been using our implementation as an experiment in software engineering. In [Kerr82] we described the software engineering techniques that we have employed. In [He82], we gave a preliminary report on how the techniques were useful. In this report we make some further observations about these techniques based on our experience during our system integration. These observations are included in Chapter 5.

2. MESSAGE PASSING IN THE MULTI-BACKEND DATABASE SYSTEM (MDBS)

In this chapter we describe the message passing functions of MDBS. The detailed description provided here will be in four parts. The first three sections will detail the different message passing mechanisms provided by the operating systems. These provide for message passing within a backend (a PDP-11/44), message passing within the controller (a VAX-11/780) and communication between the controller and the backends. The fourth section will describe the types of messages used in MDBS and their flow.

2.1. Message Passing Within a Backend

As stated earlier the backends include PDP-11/44s running the RSX-11M operating system. Under RSX-11M the inter-process communication facility available to us is the shared access to the physical memory [DEC79b]. Suppose process A wants to send a message to process B. First, A copies the message into a shared area of the memory. Then A asks the operating system to send a pointer to this area to process B. The operating system maintains a queue of pointers for process B. When B is ready to use a message, it gets the pointer from the operating system and then copies the message into its own memory.

The following three definitions provide a basis for Figures 3a and 3b which show more precisely how the above concept is implemented.

* Physical Address Space - a computer's physical address space consists of the entire addressable physical memory. In our case, the space is of 256K bytes.

* Logical Address Space - a process's logical address space is the total amount of the physical memory to which the process has access rights. This includes areas of the physical memory called regions. Each region occupies a contiguous block of memory.

* Virtual Address Space - a process's virtual address space is of 64K bytes. The process can divide its virtual address space into segments called virtual address windows. It can utilize space larger than 64K bytes using overlays.

PROCESS A	PROCESS B
1) Allocate a contiguous block of memory from the computer's physical address space. This block is known as a region and becomes part of the process' logical address space.	
2) Create a mapping from the process' virtual address space to this region. That is, map a segment of the virtual address space onto this region. The segment is called a window.	
3) Copy the message from process A into this region.	
4) Send a reference for the region to process B's receive queue.	Sending the reference by process A cause the region to be attached to process B's logical address space.
5) Eliminate process A's virtual-to-logical-address window.	Create a mapping from process B's virtual address space to this region. That is, let process B have a window to the region.
6) Detach process A's logical address space from the region.	Take the region reference from process B's reference receive queue.
7)	Copy the message from the region to process B's space.
8)	Eliminate process B's virtual-to-logical-address window.
9)	Detach process B's logical address space from the region.

Figure 3a. Conceptual flow for Process A sending a message to Process B

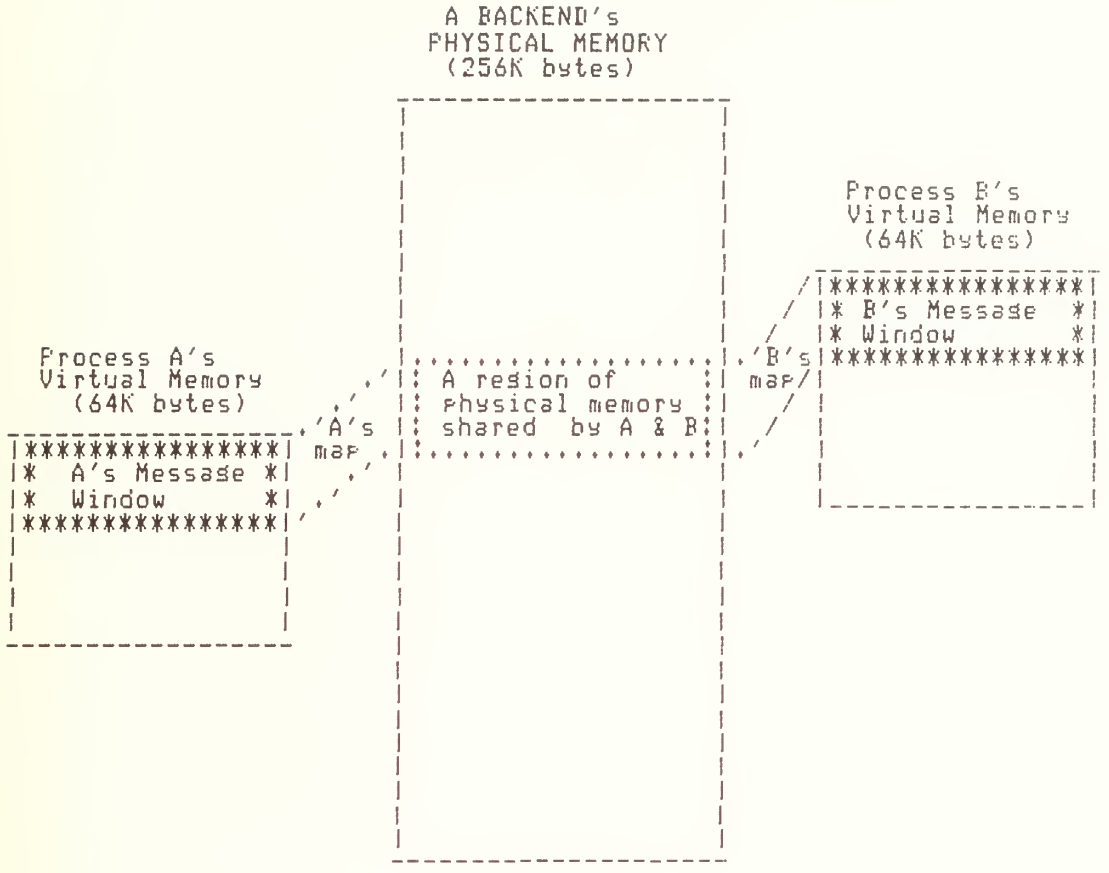


Figure 3b. Backend Message Passing

We have supplied each backend process with a SEND procedure and a RECEIVE procedure which provide the functionality shown in Figure 3a. They are called whenever the process needs to send or receive a message.

2.2. Message Passing Within the Controller

The MDBS controller is a VAX-11/780 running the VMS operating system. For VMS the inter-process communication facility is a virtual device known as a "mailbox" [DEC80]. A mailbox is a software implemented I/O device which can perform read and write operations. Again let us examine how process A sends a message to process B (see Figures 4a and 4b). First process A sends a message to process B's mailbox. When process B issues a read on its mailbox it will be given the message sent by process A. Thus the mailbox will act as a receive queue. Messages remain in the mailbox until they are accepted by the receiving process. We note that the mailbox facility is much easier to use for inter-process communication than the RSX-11M shared memory feature, as is shown by comparing Figures 4a and 4b with Figures 3a and 3b.

We programmed functions for sending and receiving messages using mailboxes and made them available to each process in the controller. They are called whenever a process needs to send or receive a message.

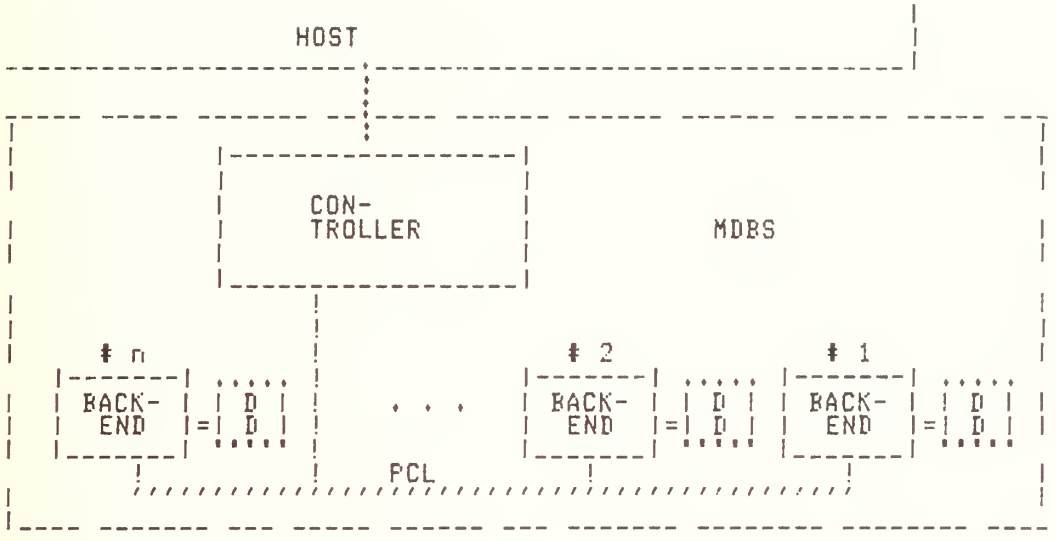
2.3. Message Passing Between Computers

MDBS (as shown in Figure 5a) needs three types of inter-computer communication. First the controller must broadcast messages to the backends. Second a backend must broadcast a message to the other backends. Finally, each backend must be able to send messages to the controller. The third case is the general function of one machine sending a message to another machine and is provided for by the hardware of our implementation. The broadcast capability is not available in the hardware. Therefore, we simulated the broadcast feature necessary for cases one and two above. Communication between computers in MDBS is achieved by using a time-division-multiplexed bus called the parallel communication link (PCL) [DEC79a]. We built a software interface to this bus for each computer consisting of two complimentary processes. The first process, get pcl, gets messages from other computers off the PCL. The second process, put pcl, puts messages on the bus to be sent to other computers.

PROCESS A	PROCESS B
1) Send message to process B	Process A's send command queues the message in process B's mailbox.
2)	Read the next message from the mailbox queue.

Figure 4a. Vax Inter-process Communication Using Mailboxes

Figure 4b. Controller Message Passing



MDBS Overview - Inter-machine Communication

1) Broadcast: Controller to backends
   A backend to other backends

2) Directed: Backend to Controller

Figure 5a. MDBS Message Passing: the Inter-machine View

The PCL allows messages from separate computers to be passed over the bus during the same period of time. Using a polling sequence each computer is given a time slice to send one 16 bit word over the bus. Over a period of time a message of length n words will be sent over the bus sequentially as n messages for sending and receiving. However, the put and get processes consider these n words as a single complete message. There is no broadcast capability built into the PCL. A computer sending a message to several other computers will have to explicitly send the message to each computer in sequence.

As mentioned above there is a get pcl process as part of the PCL interface. There can be only one process in each computer which gets messages from the PCL. This process needs to declare to the PCL that it is sole receiver of messages coming to the resident comptuer over the PCL. The get pcl process is responsible for receiving messages sent to its computer and sending them along to the specified MDBS processes within the computer.

The put pcl process of the PCL interface, transmits messages over the PCL to some other computers' get pcl processes. If the put pcl process is in the controller then it will transmit a message to each backend (i.e., broadcast). In a backend there are two possibilities. The first is that a message is to be sent to the controller, this means a single transmission. Secondly, if the message contains descriptor id's then it will be sequentially transmitted to all the other backends. If a transmission by a put pcl process fails to be accepted by the destination computer, then the sending process will wait and try the transmission later. For example, if computer A sends a message to computer B, while computer C is already sending a message to computer B. Then computer A's message will not be acknowledged by the get pcl process in computer B. Therefore, the put pcl process in computer A will wait a certain amount of time and retransmit the message. Figure 5b diagrams the sending of a message from computer A to computer B.

2.4. MDBS Message Descriptions

Among the reasons for implementing MDBS with a message-oriented approach was the facilities that VMS and RSX-11M supply for message passing. There are 23 message types defined in MDBS and one general message format. The standard message form used by MDBS is shown in Figure 6. This same format is used for each of the three message passing methods, namely, reference, mailbox and PCL.

MACHINE A ('put' process)	MACHINE B ('get' process)
1) Receive a message from an MDBS process in machine A.	
2) Put the message on the PCL for machine B.	
3)	Get the message sent by machine A off the bus.
4)	Send the message (from machine A) to the specified process in machine B.

Figure 5b. Message Passing from Machine A to Machine B

Message type	(a numeric code),
Message Sender	(a numeric code),
Message Receiver	(a numeric code),
Message Text	(an alphanumeric field terminated by an end-of-message marker),

Figure 6. MDBS General Message Format

Figure 7 describes each of the MDBS message types, how many of the messages use the PCL processes, how many of them use only the mailboxes and how many message types are only used in the backends.

Any message passing between computers will use the PCL processes get pcl and put pcl. For this discussion we will assume this information and not mention the use of the PCL processes. For instance when the controller sends to a backend a parsed traffic unit we will not mention the intermediate destination in the message's flow as VAX put pcl and RSX get pcl processes. Therefore, excluding the PCL interface processes there are three processes in the controller and three processes in each backend. Figure 8 depicts the MDBS system, showing the six processes.

2.4.1. MDBS Message Definitions

In the following we give short descriptions for messages used in MDBS. The first group of messages are those between the host and the controller and within the controller itself. These messages are shown in Figure 9.

Message type : (1) Traffic unit
Source : Host
Destination : Request Preparation
Explanation : The traffic unit represents a single request or transaction from a user at the host machine.

Message type : (2) Results
Source : Post Processing
Destination : Host
Explanation : Contains the results corresponding to a traffic unit after being collected from all the backends and aggregated if necessary.

Message type : (3) Number of requests in a transaction
Source : Request Preparation
Destination : Post Processing
Explanation : Request Preparation sends to Post Processing the number of requests in a traffic unit. This enables Post Processing to determine whether the processing of a traffic unit is complete.

Message type : (4) Aggregate Operators
Source : Request Preparation
Destination : Post Processing
Explanation : Request Preparation sends the aggregate operators to Post Processing.

MDBS MESSAGES:

No.	TYPE	SRC	DEST	PATH
1	TRAFFIC UNIT	1 HOST	1 REQP	1 HC
	RESULTS	PP	HOST	CH
	NUMBER OF REQUESTS IN A TRANSACTION	REQP	PP	C
	AGGREGATE OPERATORS	REQP	PP	C
5	REQUESTS WITH ERRORS	5 REQP	5 PP	5 C 5
	PARSED TRAFFIC UNIT	REQP	DM	CB
	NEW DESCRIPTOR ID	IIG	DM	CB
	BACKEND NUMBER	IIG	DM	CB
	CLUSTER ID	DM	IIG	BC
10	REQUEST FOR NEW DESCRIPTOR ID	10 DM	10 IIG	10 BC 10
	BACKEND RESULTS FOR A REQUEST	RECP	PP	BC
	BACKEND AGGREGATE OPERATOR RESULTS	RECP	PP	BC
	RECORD THAT HAS CHANGED CLUSTER	RECP	REQP	BC
	RESULTS OF A RETRIEVE OR FETCH CAUSED BY AN UPDATE	RECP	REQP	BC
15	DESCRIPTOR IDS	15 DM	15 DMs	15 BB 15
	REQUEST AND DISK ADDRESSES	DM	RECP	B
	CHANGED CLUSTER RESPONSE	DM	RECP	B
	UPDATED RECORD INSERTED	DM	RECP	B
	FETCH	DM	RECP	B
20	OLD AND NEW VALUES OF ATTRIBUTE BEING MODIFIED	20 RECP	20 DM	20 B 20
	CLUSTER IDS FOR A TRAFFIC UNIT	DM	CC	B
	REQUEST ID OK TO EXECUTE	CC	DM	B
23	REQUEST ID OF A FINISHED REQUEST	23 RECP	23 CC	23 B 23

SRC / DEST		PATH	
HOST	: HOST MACHINE (TEST-INT)	H	: HOST
REQP	: REQUEST PREPARATION	C	: CONTROLLER
IIG	: INSERT INFORMATION GENERATION	B	: A BACKEND
PP	: POST PROCESSING		
DM	: DIRECTORY MANAGEMENT		
RECP	: RECORD PROCESSING		
CC	: CONCURRENCY CONTROL		

Figure 7. The MDBS Message Types

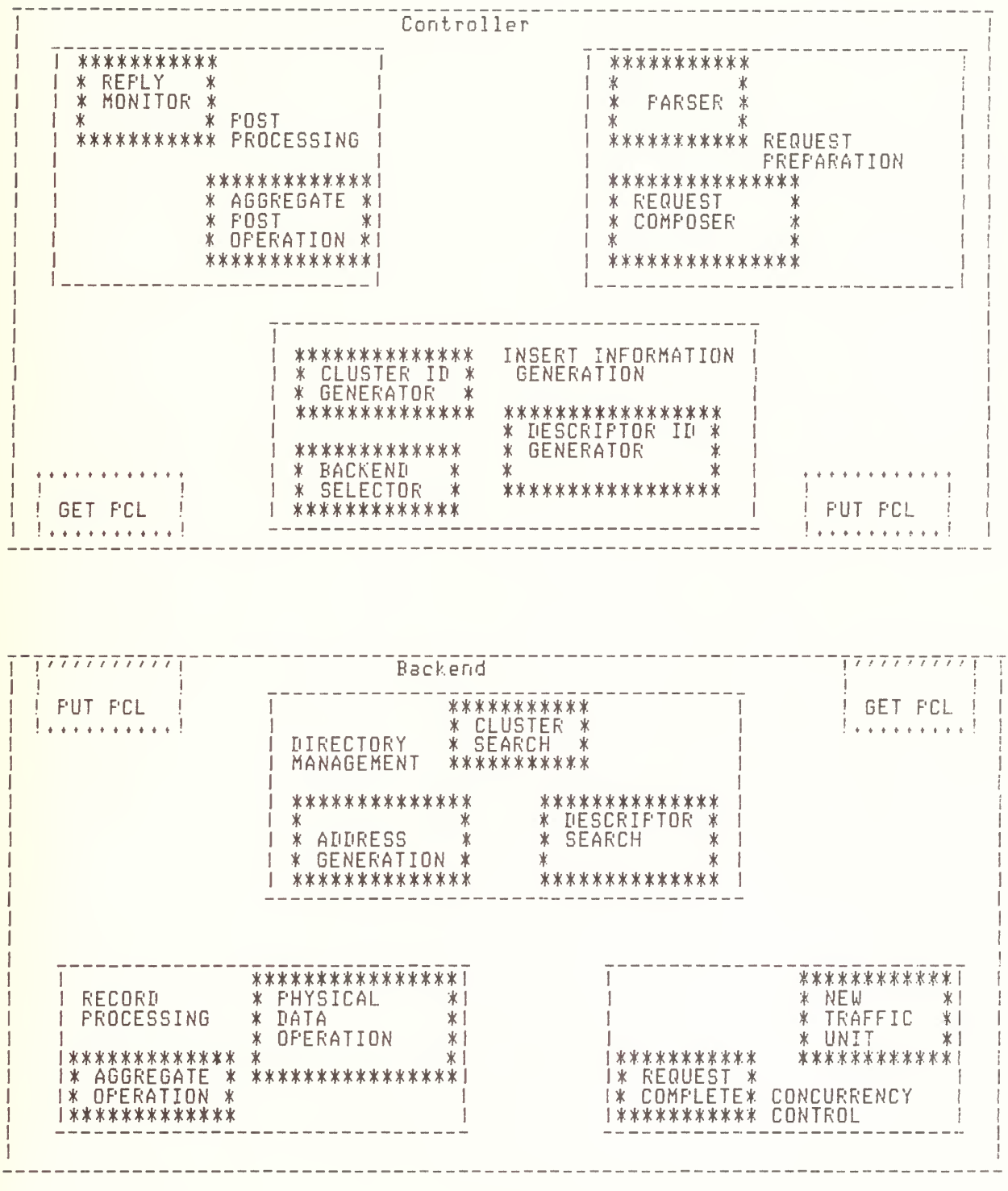


Figure 8. MDBS Controller and Backend Configuration

Figure 9. Controller Related Messages

Message type : (5) Requests with Errors
Source : Request Preparation
Destination : Post Processing
Explanation : Requests with errors will be found in Request Preparation by the Parser and sent to the Post Processing directly. Post Processing will send the requests with errors back to the host.

The next set of messages deals with the communication between the controller and the Directory Management process within each backend. These messages can be found on Figure 10.

Message type : (6) Parsed Traffic Unit
Source : Request Preparation
Destination : Directory Management
Explanation : This is the prepared traffic unit sent by Request Preparation.

Message type : (7) New Descriptor Id
Source : Insert Information Generation
Destination : Directory Management
Explanation : This message is a response to the Directory Management request for a new descriptor id.

Message type : (8) Backend Number
Source : Insert Information Generation
Destination : Directory Management
Explanation : This message is used to specify which backend is to insert a record.

Message type : (9) Cluster Id
Source : Directory Management
Destination : Insert Information Generation
Explanation : Directory Management sends a cluster id to Insert Information Generation for an insert request. IIG will decide where to do the insert.

Message type : (10) Request for New Descriptor Id
Source : Directory Management
Destination : Insert Information Generation
Explanation : When Directory Management has found a new descriptor it is sent to Insert Information Generation to generate an id.

The third group of messages deal with the flow from the Record Processing process in a backend to the Post Processing and Request Preparation processes in the controller. Figure 11 shows the flow of these messages.

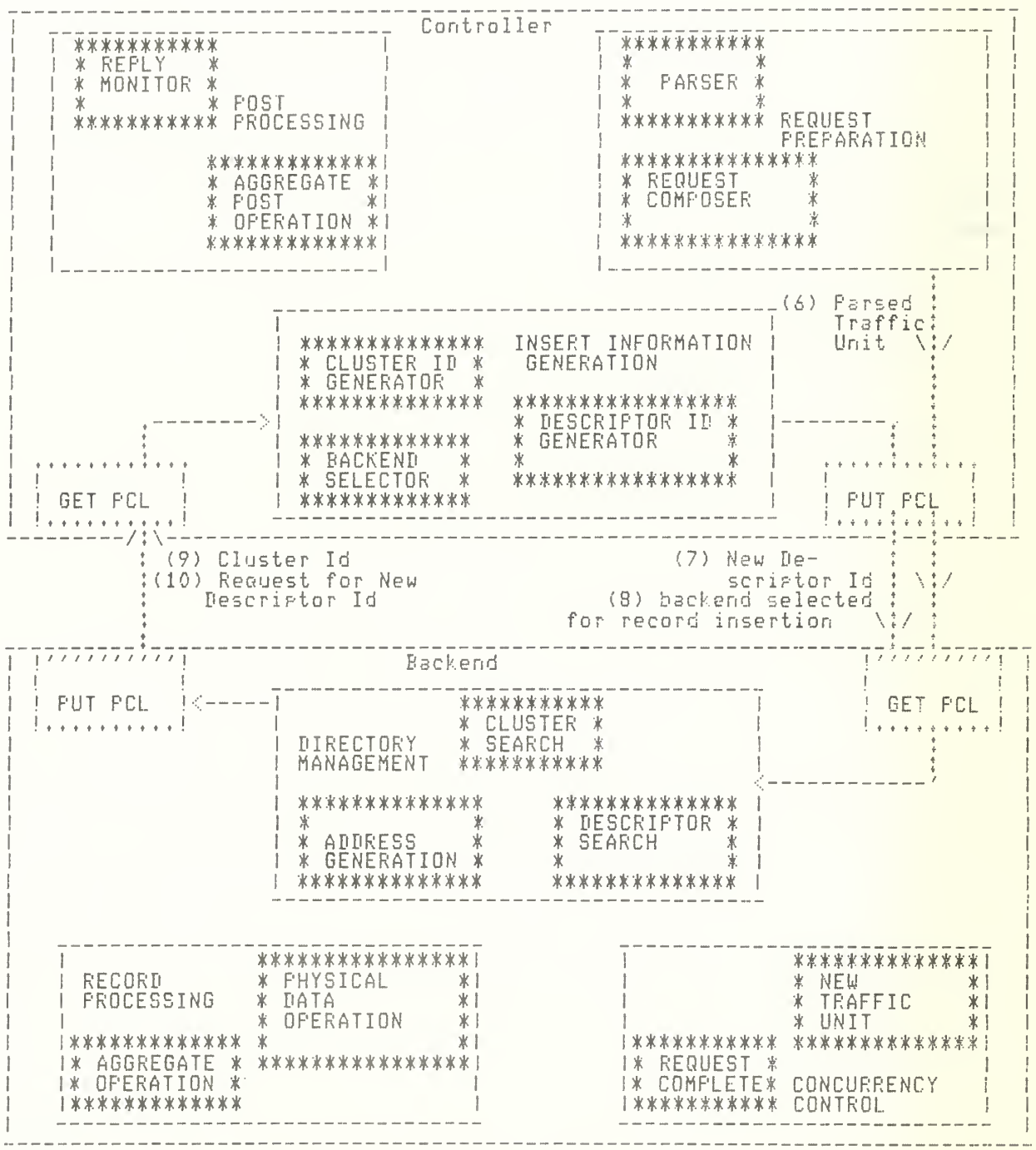


Figure 10. REQP, IIG (Controller); DM (Backend) Related Messages

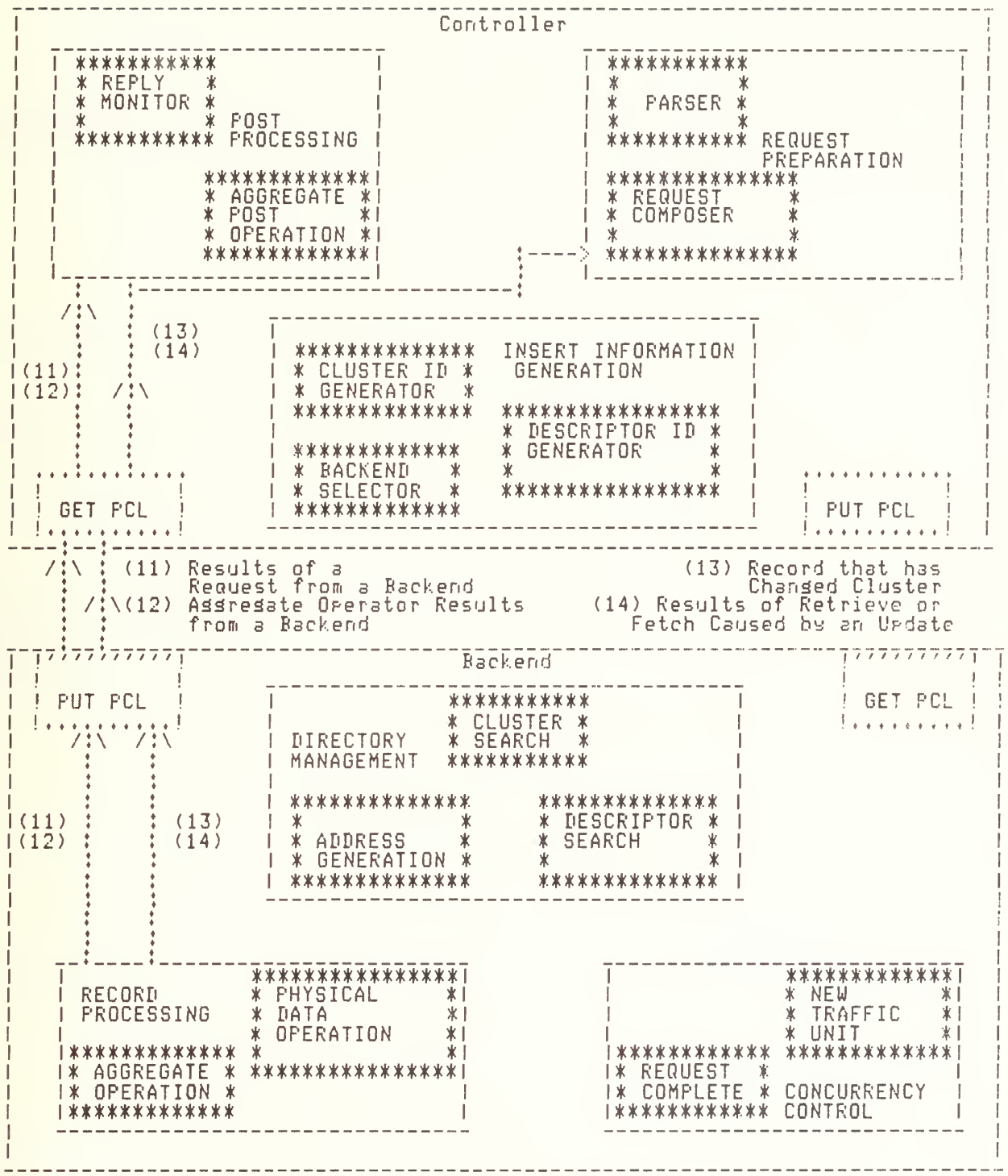


Figure 11. REQP, RECP and PP Related Messages

Message type : (11) Results of a Request from a Backend
Source : Record Processing
Destination : Post Processing
Explanation : This message contains the results that a specific backend found for a request.

Message type : (12) Aggregate Operator Results from a Backend
Source : Record Processing
Destination : Post Processing
Explanation : When an aggregate operation needs to be done on the retrieved records, each backend will do as much aggregation as possible in the aggregate operation function of Record Processing. This message carries those results to Post Processing.

Message type : (13) Record That Has Changed Cluster
Source : Record Processing
Destination : Request Preparation
Explanation : This message is a record which has changed cluster, Request Preparation will prepare it as an insertion and send it to the backends.

Message type : (14) Results of a Retrieve or Fetch Caused by an Update
Source : Record Processing
Destination : Request Preparation
Explanation : This message carries the information from a retrieve or fetch back to Request Preparation to complete an update with type-III or type-IV modifier.

The following descriptions are for messages between Directory Management processes residing on different backends and between Directory Management and Record Processing within a backend. These messages are shown in Figure 12.

Message type : (15) Descriptor Ids
Source : Directory Management
Destination : Directory Management (other backends)
Explanation : This message contains the results of descriptor search by Directory Management.

Message type : (16) Request and Disk Addresses
Source : Directory Management
Destination : Record Processing
Explanation : This message contains a request and disk addresses for Record Processing to come up with the results for the request.

Message type : (17) Changed Cluster Response
Source : Directory Management
Destination : Record Processing
Explanation : Directory Management uses this message to tell Record Processing whether an updated record has changed cluster.

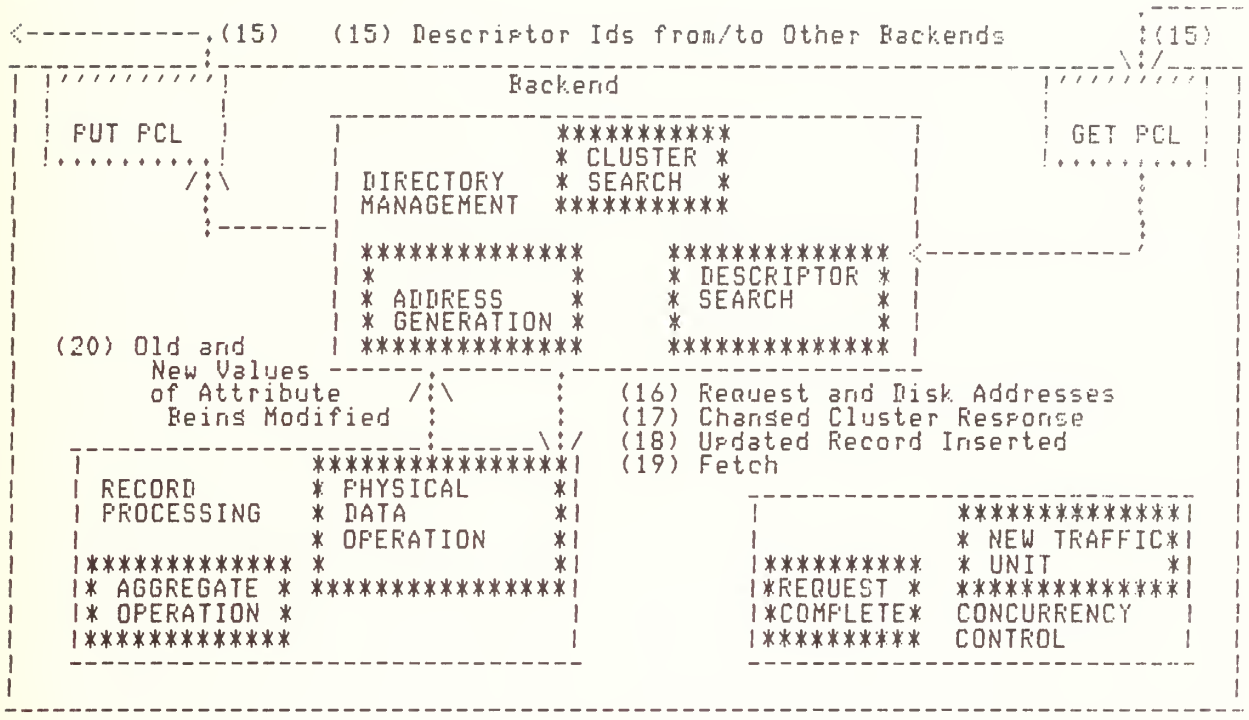


Figure 12. (Backend) DM and RECP Related Messages

Message type : (18) Updated Record Inserted
Source : Directory Management
Destination : Record Processing
Explanation : Directory Management uses this message to tell Record Processing that a record sent to Request Preparation due to an update has been inserted at a backend. Record Processing needs this information to determine whether the processing of the original update is complete.

Message type : (19) Fetch
Source : Directory Management
Destination : Record Processing
Explanation : Fetch is a special retrieval of information for Request Preparation due to an update request with type-IV modifier.

Message Type : (20) Old and New Values of Attribute being Modified
Source : Record Processing
Destination : Directory Management
Explanation : Record Processing uses this message to check whether a record that has been updated has changed cluster.

The last set of messages are the Concurrency Control related messages. These messages pass information between Directory Management, Concurrency Control and Record Processing. These are shown in Figure 13.

Message Type : (21) Cluster Ids for a Traffic Unit
Source : Directory Management
Destination : Concurrency Control
Explanation : Concurrency Control takes the cluster ids in this message and determines when the requests in the traffic unit can execute.

Message Type : (22) Request Id Ok to Execute
Source : Concurrency Control
Destination : Directory Management
Explanation : Concurrency Control tells Directory Management which request can now execute.

Message Type : (23) Request Id of a Finished Request
Source : Record Processing
Destination : Concurrency Control
Explanation : Record Processing tells Concurrency Control which request has just completed.

2.4.2. Request Execution in MDBS - Viewed via Message Passing

In this section, we describe the sequence of actions taken by MDBS in executing each of the four types of requests: insert, delete, retrieve and update. The sequence of actions will be described in terms of the types of messages passed between the MDBS processes: Request Preparation (REQP), Insert Information Generation (IIG), Post Processing (PP), Directory Management (DM),

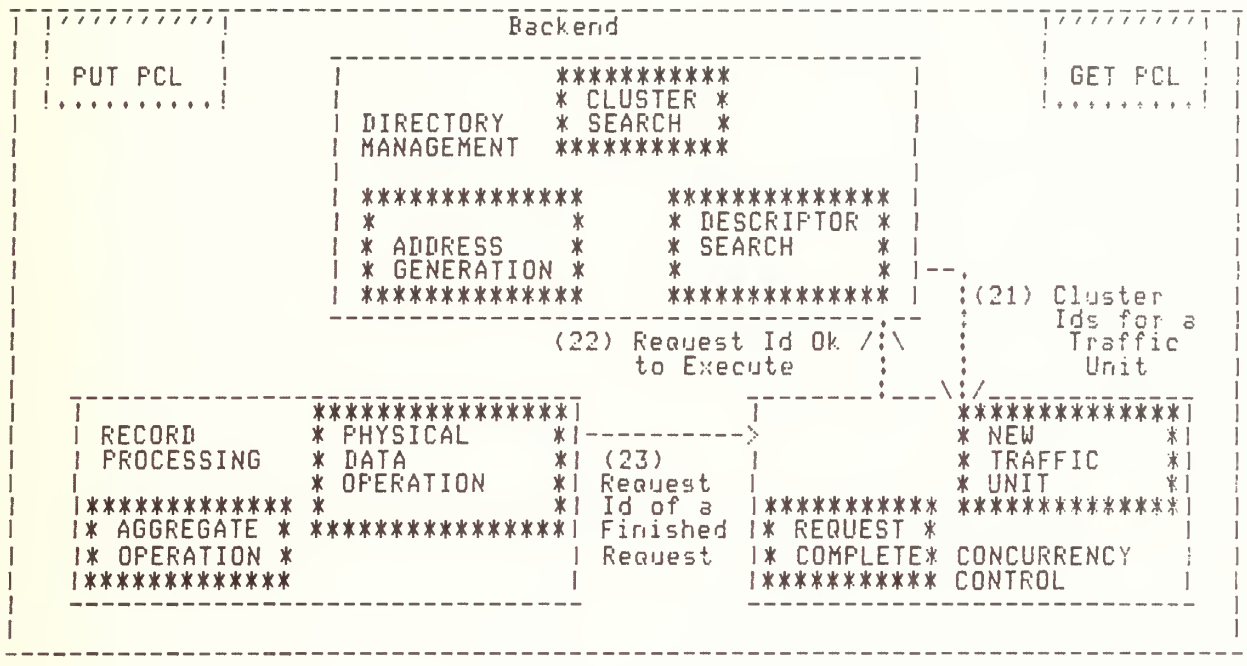


Figure 13. (Backend) DM, RECP and CC Related Messages

Record Processing (RECP) and Concurrency Control (CC). The order in which messages are passed will be denoted alphabetically ('a' is first). The digit following the ordering letter will be the message number as shown in Figure 7.

(A) Sequence of Actions for an Insert Request

The sequence of actions for an insert request is shown in Figure 14. The traffic unit (a1) comes into REQP from the host carrying an insert request. REQP sends to PP the number of requests in the transaction (b3). After preparation, the formatted request is sent to DM from REQP (c6). From the DM, descriptor ids for the request will be sent to the other backends in the MDBS system (d15). The descriptor ids found by the other backends will then be received by DM (e15). To determine where the insert will occur DM will send the insert cluster id to IIG (f9). Once the backend has been selected, IIG will send the backend number to DM (g8). Concurrency Control must determine if the insert can proceed; therefore, DM will send the insert cluster id to CC (h21). CC will respond to DM with the request id of an executable request (i22). With the go ahead from CC, DM will send RECP the request and its required disk address (j16). After the insert has occurred, RECP will notify CC that the request is done (k23), followed by a message to PP that the transaction has completed (l11). PP will finish the processing by sending a results message to the host (m2).

(B) Sequence of Actions for a Delete Request

The sequence of actions for a delete request is shown in Figure 15. A traffic unit is sent to REQP from host containing the delete request (a1). REQP notifies PP of the number of requests in the transaction (b3). Next, REQP sends the request down to DM (c6). The descriptor ids for the request are next sent to the other backends from DM (d15). The other backends respond with the descriptor ids they have found (e15). DM will next send the cluster ids to CC to assure the delete can go through (f21). CC responds to DM with the id of the next executable request (g22). Next, RECP receives the addresses and request from DM (h16). After RECP has performed the delete request it will notify CC that the request is through (i23). PP will then receive a results message from RECP telling it that the request is done (j11). PP will then notify the host with a results message (k2).

(C) Sequence of Actions for a Retrieve Request with Aggregate Operator

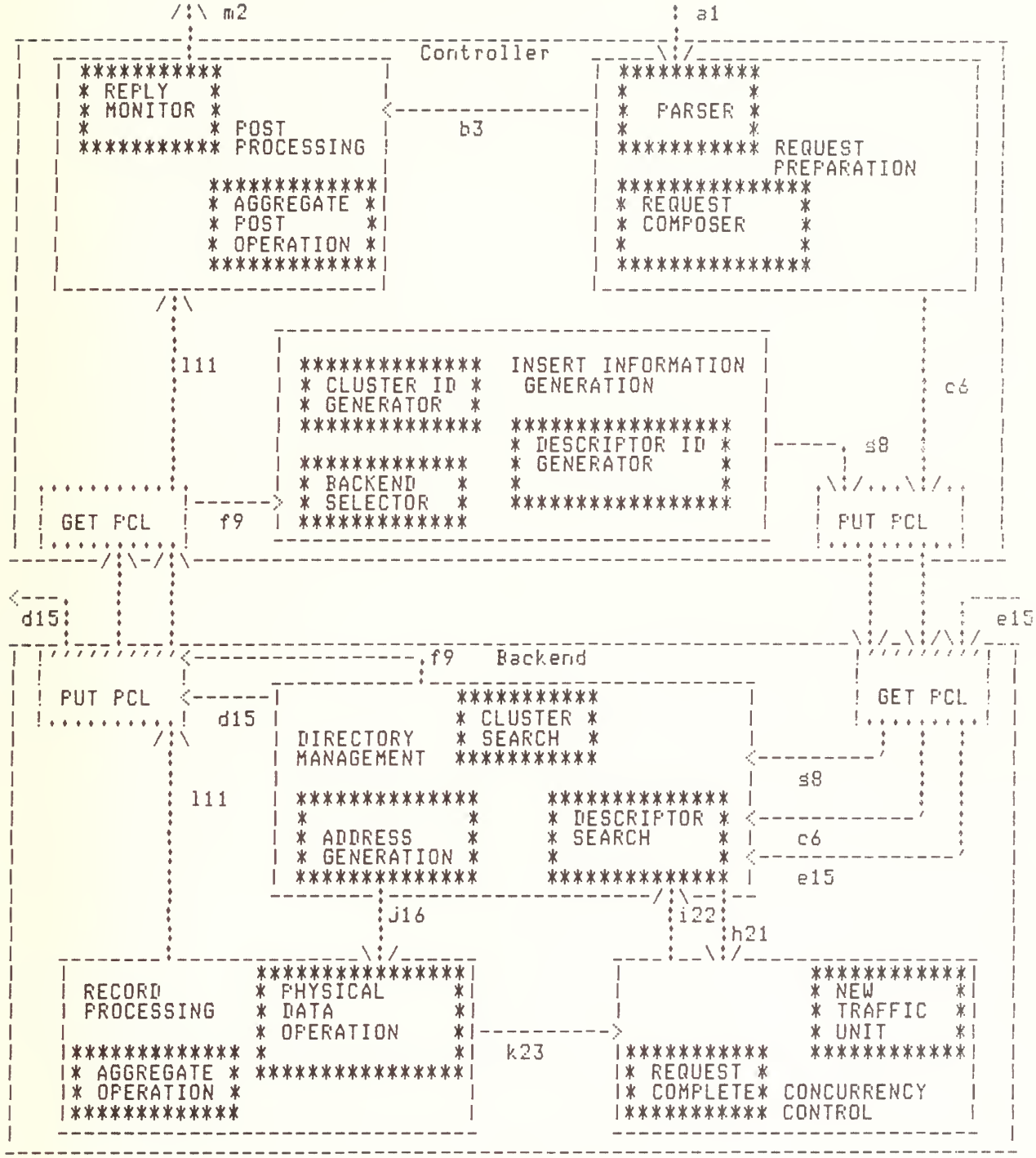


Figure 14. Sequence of Message Passing Events for an Insert Request

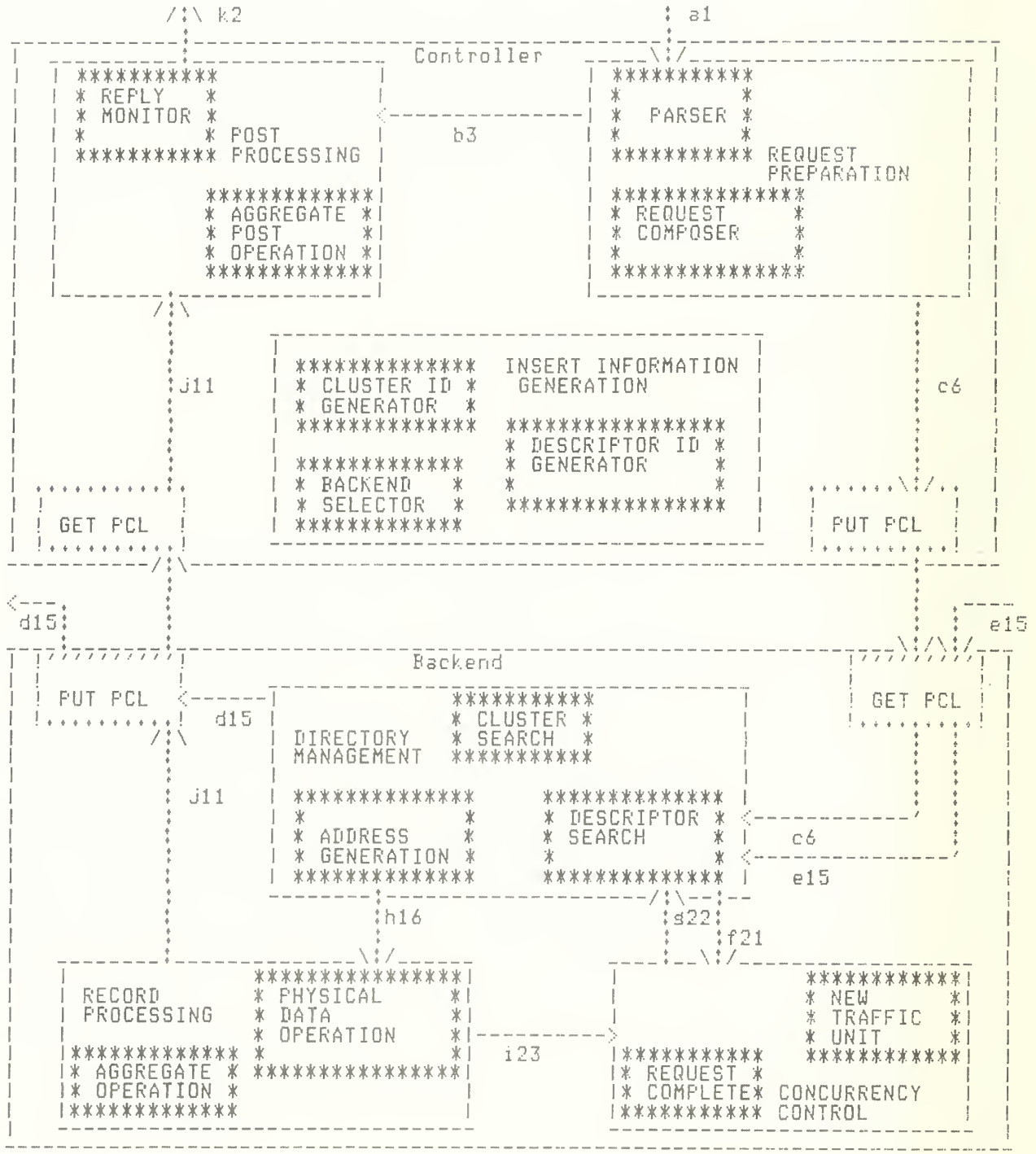


Figure 15. Sequence of Message Passing Events for a Delete Request

The sequence of actions for a retrieve request is shown in Figure 16. First the retrieve request comes to the controller REQP from the host (a1). REQP sends two messages to PP: the number of requests in the transaction (b3) and the aggregate operator of the request (c4). The third message sent by REQP is the parsed traffic unit which goes to DM in the backends (d6). DM will send the descriptor ids determined from the request to the other backends (e15). The DM processes in the other backends will send their descriptor ids to the DM process residing in this backend (f15). Next, DM will send the cluster ids for the retrieval to CC (g21). CC determines which request can execute next and sends that id to DM (h22). The addresses and the request are sent from DM to RECP for the retrieval (i16). Once the retrieval request has executed properly, RECP will tell CC that the request is done (j23). After the retrieval results have been aggregated within the backend, that result will be sent to PP for further aggregation (k12). When PP is done, the final results will be sent to the host (l2).

(D) Sequence of Actions for an Update Request Causing an Updated Record to Change Cluster

The sequence of actions for an update request that causes a record to change cluster is shown in Figure 17. This request is processed in two parts. First, after processing the update, it is determined that a record has changed cluster. Then, an insert is generated to actually store the new record. As in the previous examples we will go through the complete execution of this request. The host sends the update request to REQP (a1). REQP follows through by formatting the request and sending PP the number of requests in the transaction (b3). DM also receives a message from REQP, the parsed traffic unit (c6). The DM in each backend will exchange descriptor ids with each of the other backends (d15 and e15). The DM will send the cluster ids to CC to check if the request can be executed (f21). Once CC responds to DM that the request can go through (g22), DM will generate the disk addresses and send the request as well as the addresses to RECP (h16). When RECP retrieves the old values of the attribute being modified by the update, it will send these old values and the new values to DM to check for records that have changed cluster (i20). A reply will be sent to RECP from DM stating (for our example) that the update does cause a record to change cluster (j17). The change of cluster by a record requires an insert, therefore RECP will send the record that has

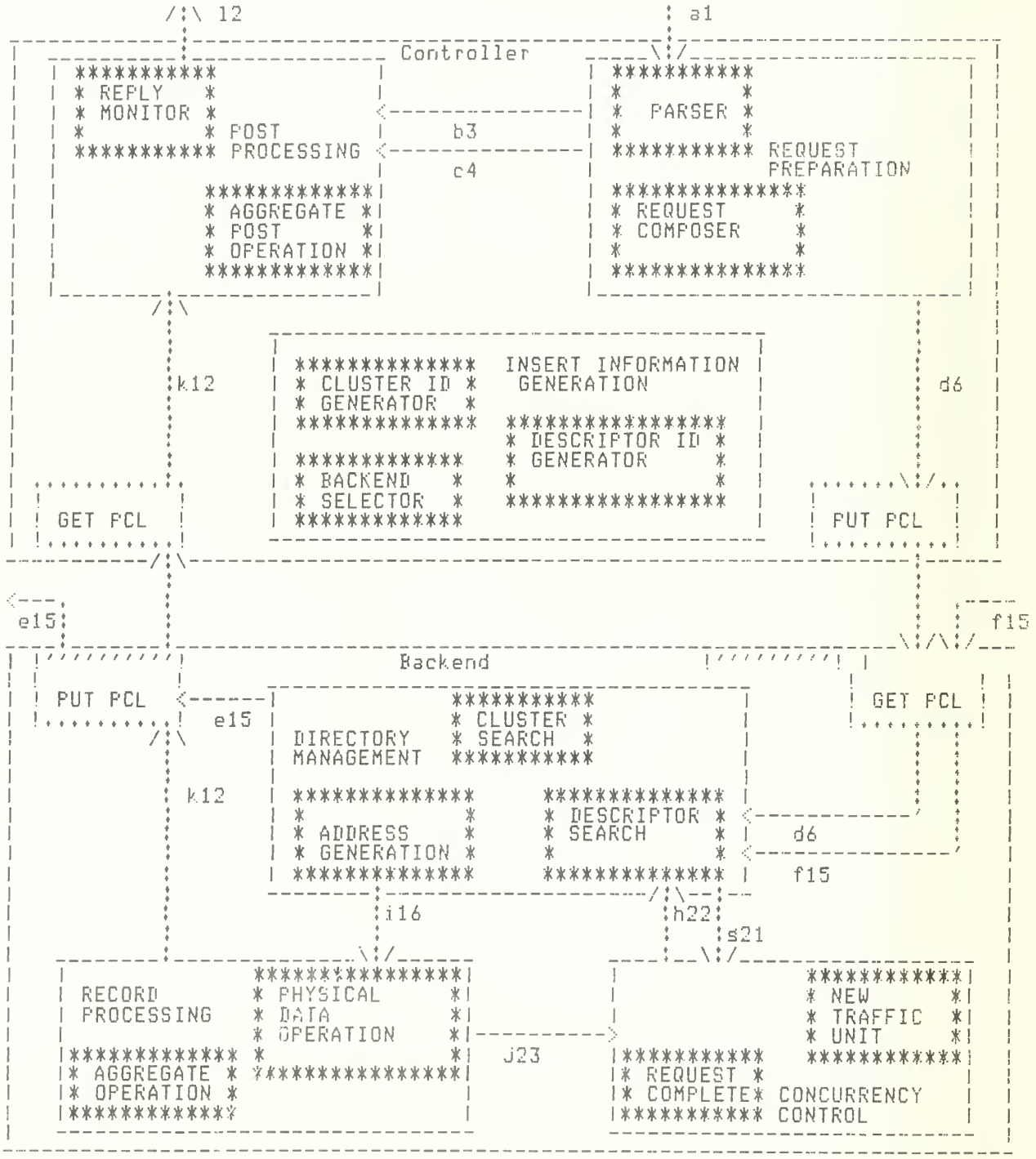


Figure 16. Sequence of Message Passing Events for a Retrieve Request With an Aggregate Operator

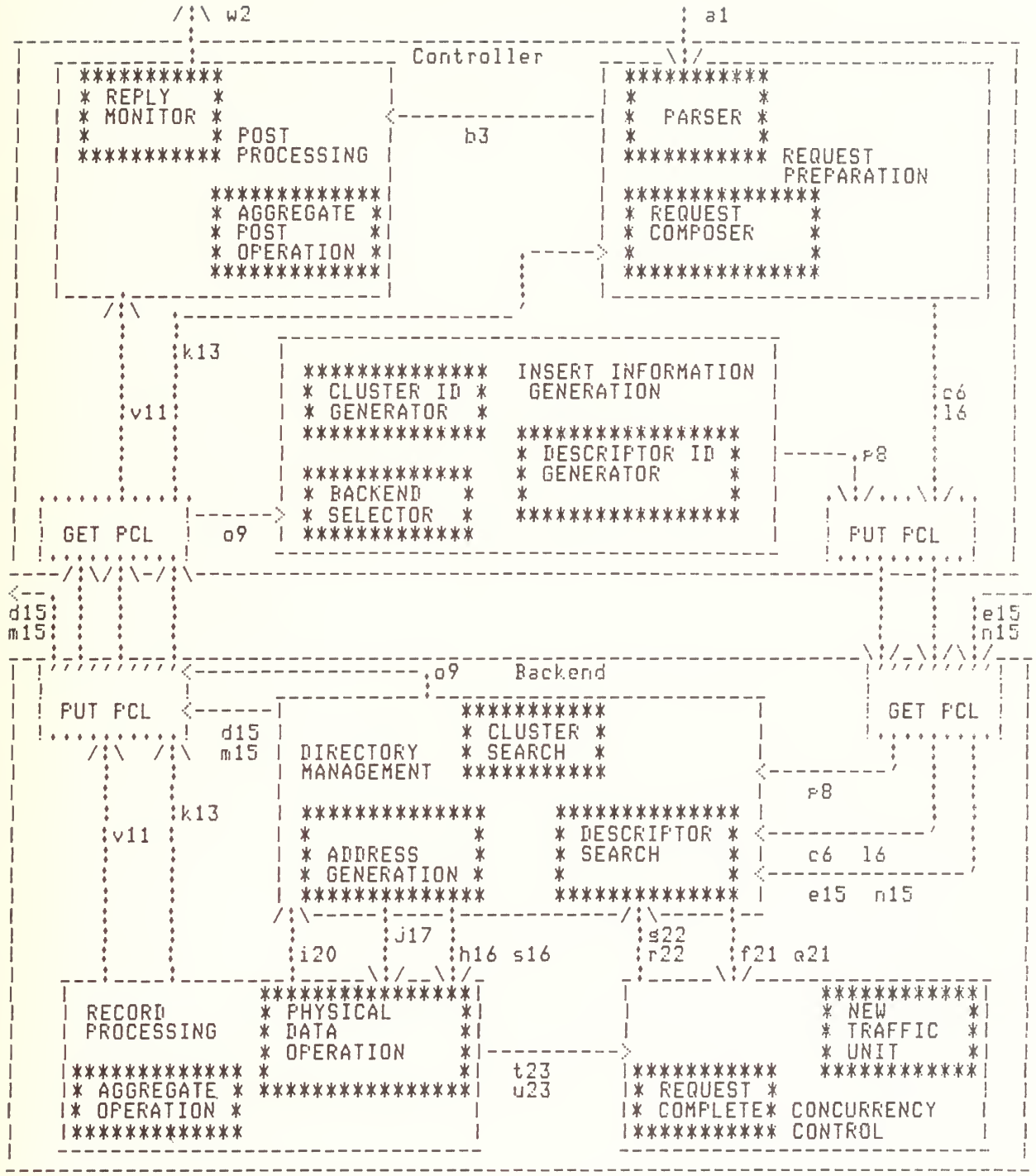


Figure 17. Sequence of Message Passing Events for an Update Request That Causes a Record to Change Cluster

changed cluster to REQP (k13). REQP will then generate an insert request. The execution of this request now proceeds as other insert requests. REQP sends DM the parsed traffic unit for the insert (l6). DM exchanges the descriptor ids for this request with the other backends (ml5 and nl5). Next, DM asks IIG where to do the insert (o9). The backend number for record insertion will be sent to the backends and DM will receive this message (p8). Before this insert can proceed from DM, CC must be given the cluster id for the request (q2l). When the request can execute, CC will send the request id to DM (r22). DM will now send the request and the necessary disk address to RECP (sl6). Once the insert completes, RECP will notify CC that the insert is done (t23). Now that the insert is completed RECP can tell CC that the update request is finished (u23). RECP next sends the results of the update request to PP (vll) and PP notifies the host that the update has completed (w2).

3. CONCURRENCY CONTROL

The MDBS concurrency control algorithm was first described in [Hsia81b]. An introduction to the implementation was given in [He82]. In this chapter we review the basic algorithm and then describe the implementation in more detail. We have made a few changes in the original algorithm, which are also described here.

Concurrency control is a mechanism by which we will insure the consistency of the database while allowing concurrent execution of multiple requests. To insure the consistency of the data, locks are utilized. These locks are administered at the cluster level (i.e., individual clusters are locked). There are five phases in the execution of a request in the presence of access control and concurrency control. First, directory management determines the clusters needed by the request. Second, cluster access control determines the authorized clusters. Third, concurrency control determines when all authorized clusters needed by the request are available. Fourth, address generation determines the record addresses. Finally, record processing actually executes the request.

3.1. Two Types of Consistency

The MDBS Concurrency Control mechanism differs from others in the types of locks as well as in their utilization. The mechanism distinguishes the four types of requests (Update, Retrieve, Insert, and Delete) and utilizes a different lock mode for each type.

There are two types of consistencies which must be assured. The first type of consistency is called inter-consistency. One example of the type of problem we are concerned with is two concurrent updates of a record, which might result in the loss of one of the updates. This problem must be considered in both single- and multiple-backend systems. To preserve inter-consistency, non-concurrent execution must be assured among requests which may have different results when executed simultaneously. Requests which may execute concurrently are called compatible requests. The compatibility of two requests depends on the mode of access, e.g., two retrieve requests are compatible whereas two update requests may not. When considering a new request, if the mode of the new request is not compatible with that of one of the earlier requests which is executing, then the execution of the new request must be

delayed. Thus the MDBS concurrency mechanism locks clusters so that only compatible requests can be using a cluster at the same time.

As just described, requests are executed at the backends in the order they are received from the controller. Sometimes for performance reasons, however, it may be desirable to permute the order of execution of two requests that are not compatible. For example, suppose a sequence of three requests R1, R2 and R3 are received and R1 requires cluster C1, R2 requires clusters C1 and C2, while R3 requires cluster C2. We note in this example that R1 and R3 are compatible whereas R2 and R3 are not. In a single backend system, it would be possible to permute the execution of requests R2 and R3, allowing R3 to execute concurrently with R1 since R1 and R3 require different clusters. In order to permute the order of execution of requests in a multi-backend system, however, a mechanism must be found to assure that all backends execute the requests in the same order. Otherwise, inconsistent results can again occur. Thus in a multi-backend system it is also necessary to assure intra-consistency, i.e., requests that are not compatible must execute in the same order at all backends.

A general mechanism to allow the permutation of requests that are not compatible would be complex because it would require communication among all the backends. However, a simple mechanism can be found that will handle the special case involving an insert request. The actual insertion of a new record is performed at only one backend. It is not performed at all the other backends. Therefore, if the backends are allowed to permute a non-insert request and an insert request, then the effective order of execution of the requests at all the backends is the order used by the backend which actually performs the insertion. In other words, no permutation takes place at all the other backends. In general, two requests that are not compatible are permutable if they do not have to be executed in the same order at all the backends. Thus we see that an insert request and a non-insert request are permutable and we can assure intra-consistency if we permute the execution order of only permutable requests.

The compatibility and permutability of requests can be summarized as follows:

	Delete	Insert	Update	Retrieve
Delete	C	P	N	N
Insert	P	C	P	P
Update	N	P	N	N
Retrieve	N	P	N	C

C = Compatible
P = Permutable
N = Not permutable and not compatible

This table shows that two delete requests, two insert requests or two retrieve requests are compatible because they can be executed concurrently without the possibility of developing inconsistency. It also shows that an insert request can be permuted with a non-insert request, i.e., a delete, an update or a retrieve. As was explained above, this permutability of an insert request with a non-insert request is due to the fact that the actual insert occurs at only one backend. Only the delete, update or retrieve is actually performed at all the backends. Thus the effect is the same as it would have been if all the backends executed the requests in the order used by the backend performing the insert.

The concurrency control mechanism described in [Hsia81b] assures that requests which are not permutable and not compatible are executed in the order received by the controller. Permutable requests can, however, be executed in any order at the same backend. So to keep track of all the requests, each backend maintains a queue of requests for each cluster, in the order in which the controller received the requests. Thus no later request can be executed before an earlier request that is not permutable and not compatible has been executed. In addition, no permutable requests can execute concurrently, although the order of execution can be modified. On the other hand, compatible requests can execute together.

3.2. Three Categories of Locks

Unfortunately, allowing the permutation of requests means that a new problem may now occur, the problem of starvation. It may be possible to per-

mute one request indefinitely. Thus that request will never be allowed to execute. We introduce three categories of locks: "to-be-used", "waiting" and "being-used". As soon as a request reaches a backend, it locks the clusters it needs in the "to-be-used" category. Before the request can be executed, it must convert the locks to the "being-used" category. If a request cannot obtain a "being-used" lock on a cluster, it locks the cluster in the "waiting" category. The "waiting" category of locks secures the request's claim for a "being-used" lock on the cluster. Only requests which have still locked a cluster in the "to-be-used" category are allowed to be permuted. Thus starvation can be prevented from those requests which have locked a cluster in the waiting category. Details of how this conversion of a lock from "to-be-used" to either "waiting" or "being-used" and how this mechanism allows the permutation of requests while preventing starvation are discussed below.

3.3. The Notion of Transaction

A user may wish to treat two or more requests as a transaction. Such a set of requests is known by the user to preserve the consistency of the database if it is executed alone on a database system running on a single computer. Users may want the execution of a transaction to begin before all the requests in the transaction have been provided to MDBS. In this case, we call the transaction incompletely-specified. Unfortunately, because all clusters required by the incompletely-specified transaction cannot be determined before the execution cf the transaction is to begin, there is no algorithm which allows the use of incompletely-specified transactions without sometimes having to backup one of two transactions which have been executing concurrently. Thus in MDBS, we have chosen to restrict transactions to those that are pre-specified, i.e., all the requests in a transaction must be submitted to MDBS at the same time and before the execution of any of the requests in the transaction begins. Then MDBS must convert all locks to the "being-used" category before the execution of the transaction can begin. Locks can then be released as requests in the transaction finish the execution.

In the previous section, when we discussed compatible and permutable requests, we assumed the requests were not part of a transaction. We must now reexamine these concepts in the context of transactions. Since two compatible requests have no ill affect on each other, we can still allow their concurrent execution even when one is part of a transaction. On the other hand, the order

of execution of two permutable requests where one is in a transaction does affect the result. Thus the whole transaction should be permuted, rather than one of its requests. Because of the complexity of permuting a whole transaction, we have chosen to permute only requests that are not part of a transaction. As a design decision, we extend the definition of permutable to include a check which requires that neither of the two requests being compared can be part of a transaction.

3.4. Concurrency Control Using a Message-Oriented Approach

The concurrency control mechanism was described in [Hsia81b] using a procedure-oriented approach. Thus there was to be a lock table shared by all users. In addition, transactions were deactivated when a needed cluster was locked by other requests and were activated when the needed cluster became available.

This basic mechanism must now be transformed to reflect a message-oriented approach. In this approach, as described earlier, there is a concurrency control process. This process receives messages from the directory management process (a request or a transaction to be executed) and from the record processing process (a report that a request has been executed). When the concurrency control process determines that a request is ready for execution it forwards the request to directory management. The "shared lock table" evident in the procedure-oriented approach now appears as a table internal to the concurrency control process. This table, called the cluster-to-traffic-unit table (CTUT), is described in Section 3.4.2. As a reminder, a traffic-unit can consist of either a single request or a transaction. The concept of "deactivating" a transaction is replaced by having concurrency control hold the request in a queue until it can be forwarded to directory management for execution. The algorithms for concurrency control are described in Section 3.4.4.

3.4.1. The Process Structure in the Backends

Once a message-oriented approach has been selected, it is necessary to break up the functions of each backend into processes. The most obvious choice would be to have one process per function, i.e., five processes corresponding to descriptor search, cluster search, address generation, concurrency control and record processing, respectively. (The sixth function, cluster access

control is omitted because it is not included in our initial implementation.) To minimize the amount of inter-process communication, descriptor search and cluster search were combined into one process, directory management. The address generation function must take place after the concurrency control process, since records may be added to a cluster while a request is waiting to lock the cluster. Thus it could be combined with the concurrency control process, the directory management process or the record processing process. It could also be a separate process. For the purposes of discussing the concurrency control process, it is easiest to assume that address generation is not part of the concurrency control process.

The decision was made to have the address generation function as part of the directory management process. Basically, some of the information stored by the directory management process on each request would be needed by the address generation function. Thus by including this function in the directory management process, the overhead required to transfer the information for a request to a separate process is eliminated.

3.4.2. Cluster-To-Traffic-Unit Table (CTUT)

As was described earlier, information about the locks held on each cluster is stored in the CTUT. This table contains a queue for each cluster. Each cluster queue contains an entry for each of the requests requiring that cluster. Each entry contains an identifier for the request (the traffic unit and the request-number), the MODE of access required (delete, insert, retrieve or update), and the CATEGORY of lock held ("to-be-used", "waiting" or "being-used"). A sample CTUT with four clusters is shown in Figure 18. This table contains entries for five single requests and one transaction consisting of two requests.

3.4.3. Traffic-Unit-To-Cluster Table (TUCT)

In a procedure-oriented implementation there is a process associated with each user and this process keeps track of how many locks are still to be acquired before a transaction can be executed. However, in a message-oriented implementation, there is no such process for a user. Thus this information must be maintained in a different way. The concurrency control process stores this information in a traffic-unit-to-cluster table (TUCT) and uses TUCT to determine the status of any traffic unit. This table is essentially an inverse

	Clusters	Traffic-Units			
Q	C1	TU1 I BU	TU2 I BU	TU3 U W	TU1 and TU2 are compatible and are being executed. The lock for TU3 has been converted to "waiting", since U and I are not compatible.
U E	C2	TU3 U TBU	TU4 I BU		TU3 and TU4 have been permuted.
U E	C3	TU5,R1 D BU			The first request of TU5 is being executed.
S	C4	TU5,R2 U TBU	TU6 I W		(TU5,R2) is part of a transaction, so TU6 can't permuted with it.

Ci = Cluster i

TUj = Traffic Unit j
Rk = Request k within a traffic unit

MODE of Request

D = Delete
I = Insert
R = Retrieve
U = Update

CATEGORY of Lock Held

BU = Being-Used
TBU = To-Be-Used
W = Waiting

NOTE: TU1, TU2, TU4 and (TU5,R1) are being executed.

Figure 18. A Sample of Cluster-To-Traffic-Unit Table (CTUT)

of the CTUT. It is a reference, by traffic unit, of which clusters are required for each request of the traffic unit. In addition, this table keeps track of how many requests there are in a transaction. Figure 19 shows the TUCT corresponding to the CTUT shown in Figure 18.

3.4.4. The Processing of Concurrency Control Information

The concurrency control process receives messages from the directory management and record processing processes. A message from directory management consists of a traffic unit of new request(s) to be executed and a list of cluster(s) required by each request in that traffic unit. The cluster(s) needed by each request are always received in the same numerical order, either increasing order or decreasing order. Using this scheme deadlock is prevented. A message from record processing means that execution of a request has been completed. Concurrency control sends a message to the directory management process when a request can be executed. The directory management process performs the address generation function and then sends the request to record processing for execution.

In order to handle these messages, the concurrency control process must perform four basic functions. First, when a new traffic unit is received from directory management, an initialization must be performed locking all the required clusters in the "to-be-used" category. Second, when concurrency control receives a message from record processing that execution of a request has been completed, it must remove the request from the TUCT (and CTUT) and determine the clusters that were locked by the request. Third, whenever a new request is received, concurrency control must try to convert (to the "being-used" category) as many locks on the clusters required by the new request. Whenever a request has completed execution, concurrency control must try to facilitate the execution of those requests that were blocked by the finished request. In this case, it tries to convert the locks (in the waiting category) of the blocked requests to the being-used category. Fourth, when all locks required by a request have been converted to "being-used", directory management must be notified to begin execution of the request.

(A) The Processing of a New Traffic Unit

The TUCT and CTUT tables are implemented as queues. When the concurrency control process receives a new traffic unit from directory management, it must

Traffic-Units	Requests			
TU1 (one request)	C1 I BU			executing
TU2 (one request)	C1 I BU			executing
TU3 (one request)	C1 U W	C2 U TBU		waiting for C1
TU4 (one request)	C2 I BU			executing
TU5 (two requests)	C3 D BU	C4 U TBU		first request is executing
TU6 (one request)	C4 I W			waiting for C4

compatible

permutable

TUi = Traffic Unit i
Cj = Cluster j

MODE of Request
D = Delete
I = Insert
R = Retrieve
U = Update

CATEGORY of Lock Held
BU = Being-Used
TBU = To-Be-Used
W = Waiting

Note : Requests within a transaction are executed sequentially. To execute a request within a transaction, only the locks for the specific request have to be converted to 'being-used'.

Figure 19. The Traffic-Unit-To-Cluster Table (TUCT) corresponding to the CTUT in Figure 18.

enter the information for each request of the traffic unit into both queues. For the TUCT table, a new head queue element (containing the traffic unit identification number and the number of requests within the traffic unit) is created and appended to the end of the queue. Then, for each cluster needed by a request in the traffic unit a queue element is created which consists of the request identification number, the cluster identification number, the category("to-be-used") of the lock, and the mode(either delete, insert, retrieve or update) of the request. As this information is entered into the TUCT table, it is simultaneously entered into the CTUT table. The head elements of the CTUT queue consist of all of the possible cluster identification numbers. A queue element consisting of a traffic unit identification number, a request identification number, the category ("to-be-used") of the lock, and the mode of the request is appended at the end of the cluster queue of the CTUT table corresponding to each cluster identification number needed by a new request.

(B) The Processing of a Finished Request

When the concurrency control process receives a message from record processing that a request has finished execution, concurrency control takes the traffic unit and request identification numbers and removes the request queue elements from the TUCT table. As each request queue element is removed, the corresponding queue element of the CTUT cluster queue, associated with the cluster identification number, which contains the specified traffic unit and request identification numbers, is also deleted from it's place in the CTUT queue.

(C) The Conversion of Locks

The last function of the concurrency control process involves the conversion of locks. Before a request can be sent to the directory management process, which will forward the request to record processing for execution after address generation, concurrency control must convert all of the locks of that request from either "to-be-used" or "waiting" to "being-used".

Consider that we are given a request and an ordered set of clusters that have to be locked in the "being-used" category before the request can be

executed. The conversion of locks is done one cluster at a time, starting with the first cluster needed by the request. The decision on converting a lock on a cluster is only affected by other requests that also need that cluster. Basically, there are two choices that can be made when a request is trying to lock a specific cluster. First, a request can be executed without being permuted. This implies that the request is compatible with all earlier requests in the cluster queue, and compatible with all later requests that are executing, i.e. compatible with all later requests that have the category "being-used". If a "being-used" lock is granted in this case, the request can be thought of as executing concurrently with other executing requests on the cluster locked. Second, a request can be executed by being permuted. Once again, the request is compatible with all earlier requests and compatible with all later requests whose category is "being-used". Now assume that one or more earlier requests, whose category were "to-be-used", were permutable with the request. The permutation may take place between the request and one of these earlier requests. However, by choosing a particular early request to permute and by granting a "being-used" lock to the request, concurrency control must now cause all other earlier requests that were permutable with the request to wait until the request finishes execution. In other words, none of the other permutable requests can obtain a "being-used" lock on the same cluster. At this stage, let us formalize this concept.

Consider that a request R needs to lock a cluster J. To determine whether the lock should be granted, the cluster J queue of the CTUT table is examined. The pictorial description of the cluster J queue is given below:

Cluster J : Earlier Requests <-- R --> Later Requests

The lock on the request R is convertible (to "being-used") if the following two conditions hold:

(a) For each earlier request ER, R is compatible with ER or R is permutable with ER where ER's category is "to-be-used".

(b) For each later request LR in the category "being-used", R is compatible with LR.

Now we return to the problem of how a newly received traffic unit is processed by concurrency control.

Assume that a new traffic unit has been received and the initialization functions have been performed, i.e., the information on the traffic unit has been entered into the TUCT and CTUT tables. The algorithm then attempts to convert all locks needed by the first request in the traffic unit from the "to-be-used" to the "being-used" category. If a lock on a cluster needed by the request is not convertible, then the request is deactivated, both within the traffic unit and the corresponding cluster queue, by setting the lock to "waiting". Once a new request is locked as "waiting", concurrency control processes the next message in its message queue. If the queue is empty, then concurrency control waits for a new message from either directory management or record processing.

When record processing indicates that a request has finished execution, concurrency control must remove that request from the TUCT and CTUT tables, and try to convert locks on any request(s) that were being blocked("waiting") by the finished request. Concurrency control must also try to convert locks on the next request (if any) within the traffic unit of the finished request. Converting locks on the next request of the traffic unit is identical to the procedure described above for a newly received traffic unit. Thus we will focus on converting locks for blocked requests.

When a request R has finished execution, concurrency control scans all of the cluster queues (of the CTUT table) that R is in. For each of the cluster queues, it removes the request R, and then cycles through all of the elements of the queue looking for a "waiting" request. For each "waiting" request, the algorithm finds the request's position in the TUCT table, and then tries to convert all of the remaining locks on the cluster(s) needed by the blocked request. This procedure is performed for all "waiting" request(s) blocked in all of the cluster queues that the finished request was in.

(D) Future Modifications

As originally proposed in [Hsia81b] and as now implemented, MDBS will execute one request at a time within a traffic unit. When a new traffic unit is received, concurrency control merely tries to convert locks on the first request of the traffic unit. If the first request of the traffic unit is

executing, then as described above no other request(s) of the traffic unit can convert locks until the executing request finishes. If the first request is deactivated, i.e., a cluster needed by the request is locked as "waiting", the only way concurrency control can restart the conversion of locks for the blocked request is if a request which holds the blocking cluster finishes execution.

Because of our message-oriented implementation, it is straightforward to allow the execution of multiple requests within a traffic unit. All of the information needed for running multiple requests is in the TUCT table. Multiple-request traffic units can be identified by examining the head queue elements of the TUCT table. Each head queue element contains a field which tells the number of requests within that traffic unit. For a multiple-request unit, the determination of an executing request within the traffic unit involves the checking of each cluster needed by the request. These clusters must be locked as "being-used". Once a non-executing request in a traffic unit has been identified, concurrency control would begin converting locks for the request. The lock conversion function starts at the first cluster needed by the request which has been locked as either "to-be-used" or "waiting". At this point, the existing lock conversion function would be invoked, and flow would continue as if a new request was being processed. With the message-oriented approach we are able to make this modification without developing a new algorithm.

4. THE SECONDARY-MEMORY-BASED DIRECTORY MANAGEMENT

As described in Chapter 1, MDBS is being developed in stages, i.e., several versions of MDBS are being developed. The first five versions (A, B, C, D and E) employ a primary-memory-based directory management, i.e., all the directory information is stored in the primary memory.

The directory information of MDBS with a large database may be too large to be stored in the primary memory. (The actual size of the directory management data structures depends on the number of directory attributes, descriptors and clusters.) Therefore, versions F and G of MDBS will employ a secondary-memory-based directory management.

We recall that the directory information is stored in three tables: attribute table (AT), descriptor-to-descriptor-id table (DDIT) and cluster-definition table (CDT). In the following sections, we describe the secondary-memory-based implementation of the three directory management tables.

4.1. The Attribute Table (AT)

We recall that this table contains the directory attributes and pointers to descriptors defined on them. A sample AT is depicted in Figure 20.

The B-tree structure [Come79] is used to implement the attribute table (AT). A sample AT, stored as a B-tree, is shown in Figure 21. In this example, there are at most two keys in each node of the tree. Besides the pointers used in the B-tree structure, there is a pointer associated with each key (directory attribute). The pointer associated with an attribute points to the descriptors defined on the attribute (see Figure 21 again).

In the current implementation of the secondary-memory-based attribute table, a physical track is used for each node of the B-tree. So, there are as many attributes in a node as a track can hold. A binary search is used when searching a given node of the B-tree. Let us illustrate the searching mechanism by means of an example. Consider the attribute table depicted in Figure 22, and let us assume that we are searching for the attribute NAME. The root node (node1) of the tree is searched first. Attribute NAME is not in node1.

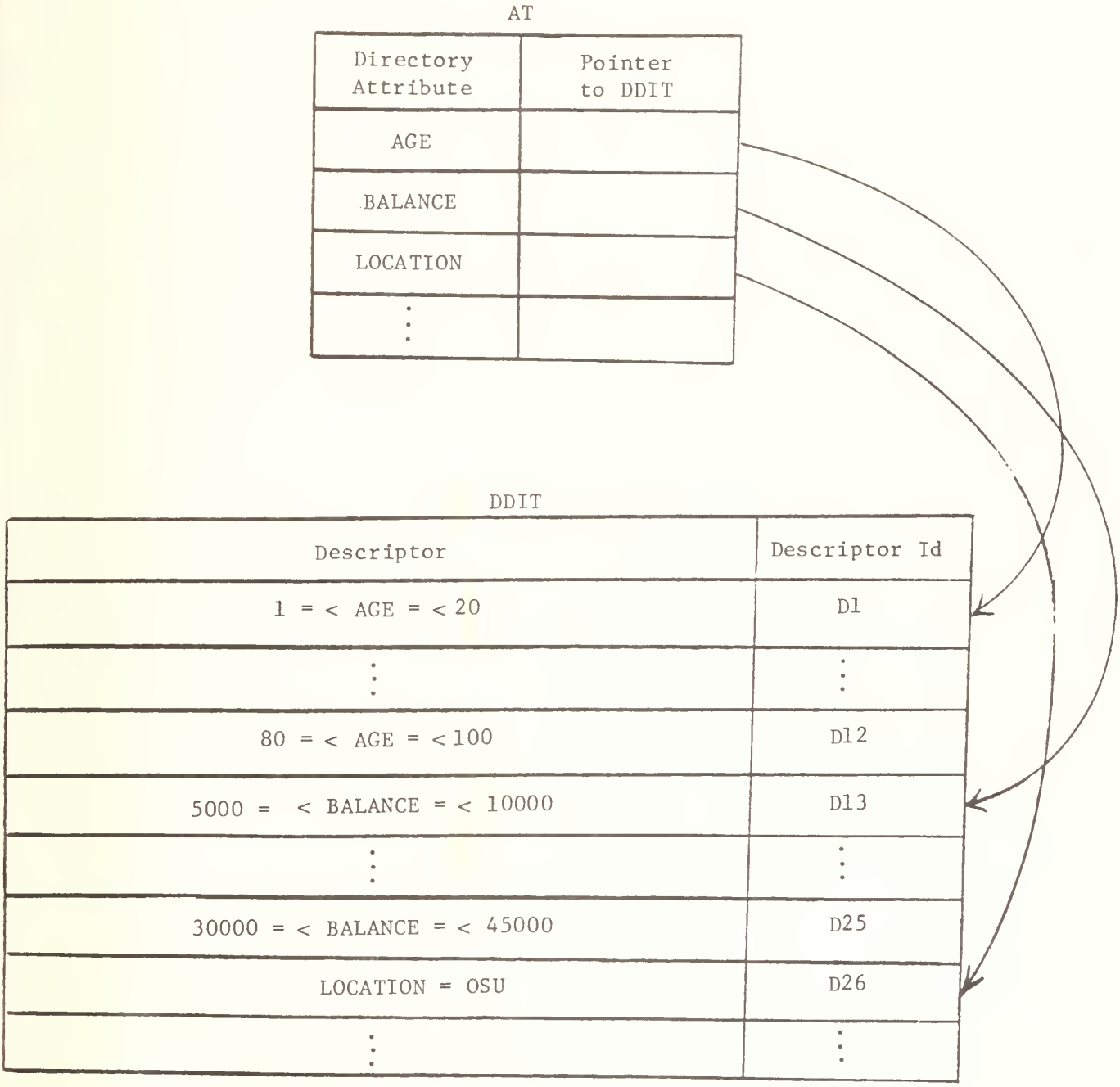
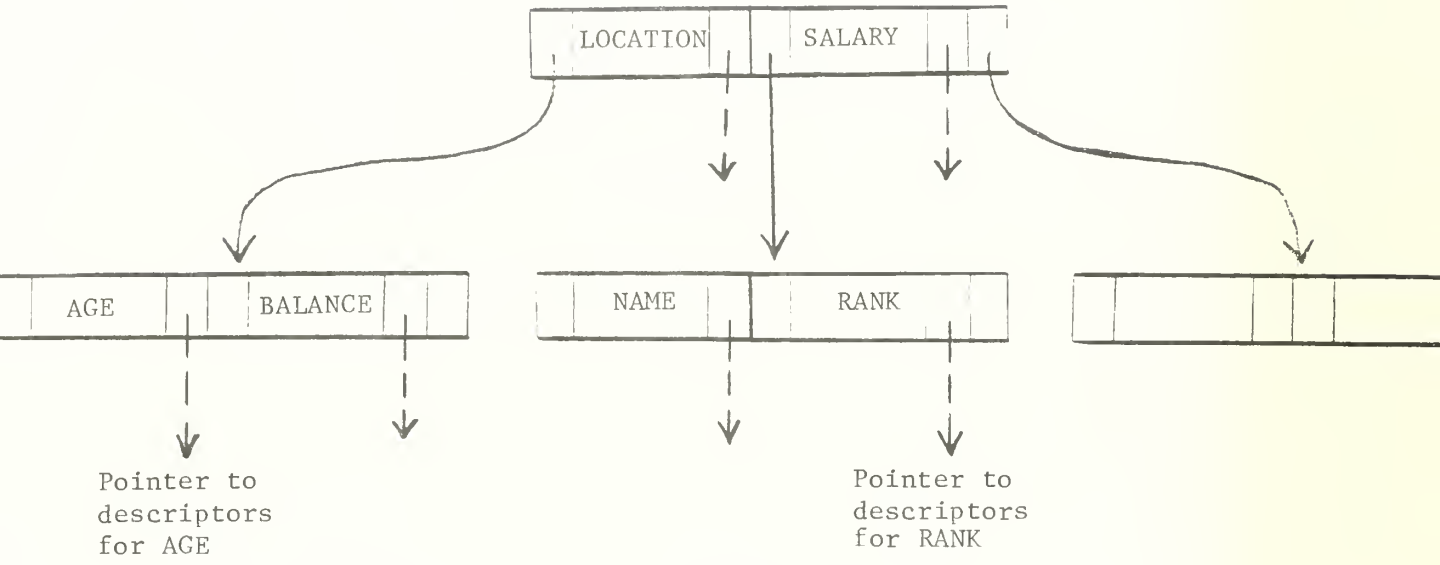


Figure 20. The Attribute Table (AT) and its Relationship to the Descriptor-To-Descriptor-Id Table (DDIT)



a. The B-tree

Ptr0	attr. name 1	Pointer to descriptors for attr name 1	Ptr 1	attr. name 2	Pointer to descriptors for attr. name 2	Ptr 2

b. The Format of a Node in the Tree

Figure 21. A Sample Attribute Table Stored as a B-Tree

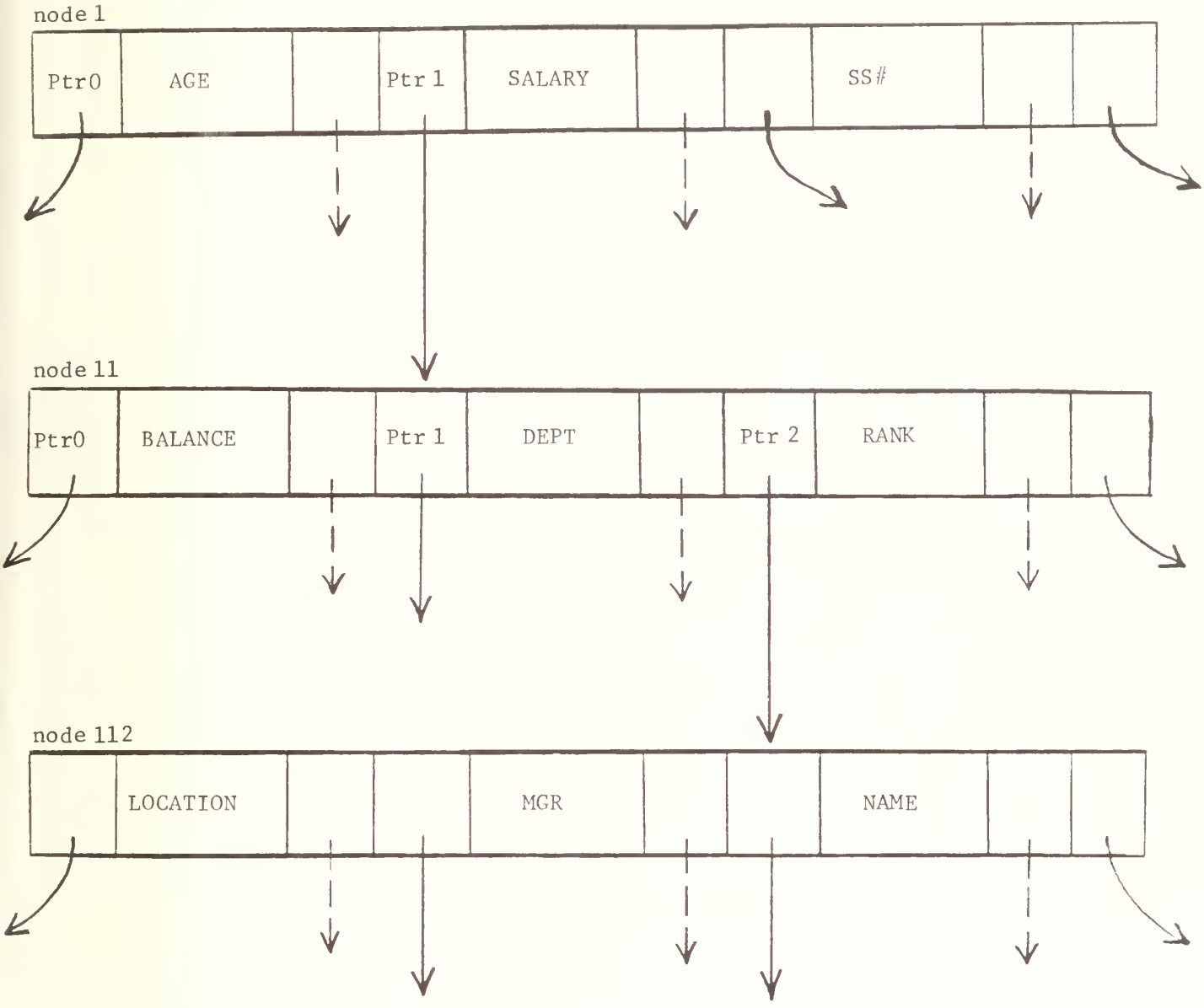


Figure 22. An Example of Look-Up in an Attribute Table

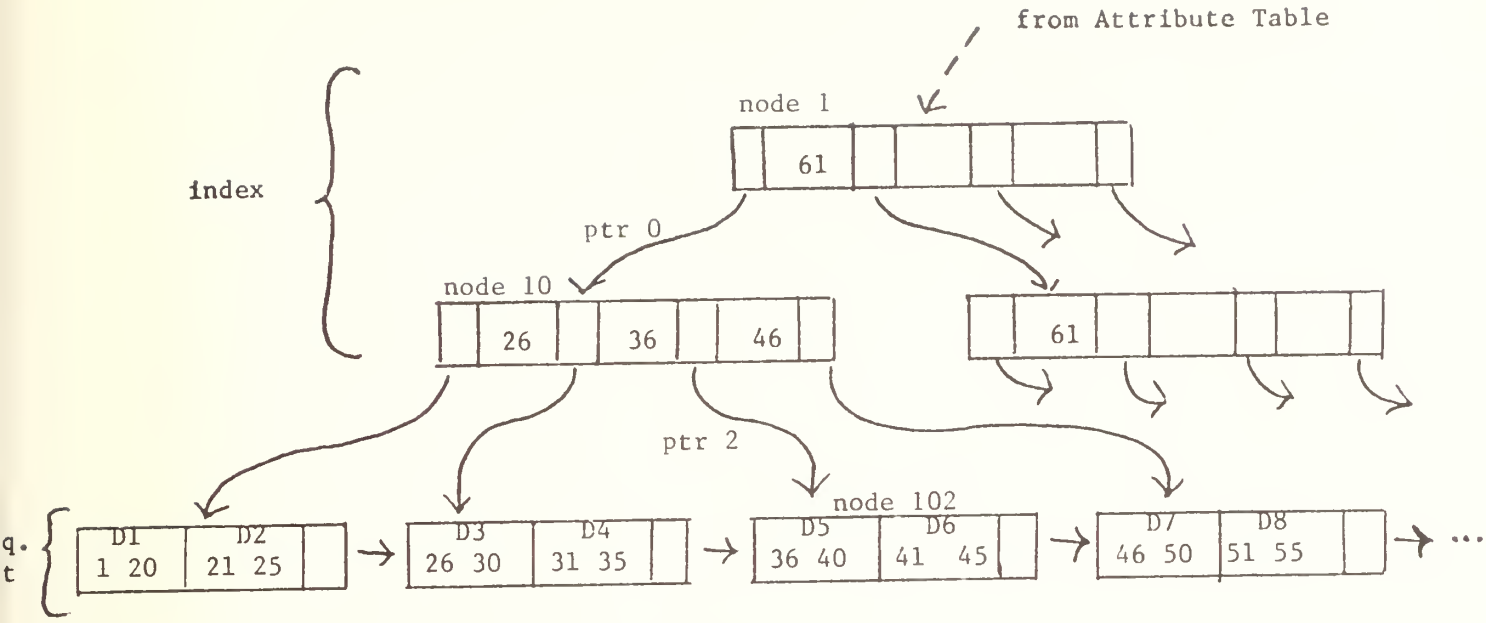
Since AGE < NAME < SAL, the subtree pointed to by ptr1 (in node1) should be searched next. Node11 is searched for the attribute NAME. Since NAME is not in node11 and DEPT < NAME < RANK, the subtree pointed to by ptr2 (in node11) should be searched next. Node112 is searched for the attribute NAME. Since NAME is in node112, the search terminates successfully.

4.2. The Descriptor-to-Descriptor-Id Table (DDIT)

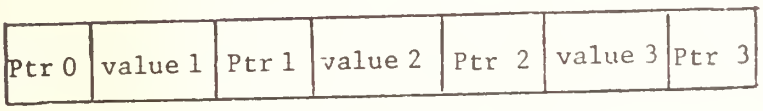
We recall that this table is used in the descriptor search phase of directory management. In this phase, the corresponding descriptor or set of descriptors for a keyword or predicate is determined. A sample DDIT was also depicted in Figure 20. Each section of this table is associated with a directory attribute and contains the descriptors defined on that attribute.

In the secondary-memory-based implementation of DDIT, a separate B+tree is used to represent the descriptors defined on a directory attribute. Thus, DDIT will be a forest of B+trees. The pointer to the B+tree for a directory attribute is stored in the attribute table. Let us explain the reason for using B+trees, rather than B-trees, by means of an example. Let us assume that we are searching for the corresponding set of descriptors for the predicate (AGE >= 42). To do this, we need to first find the corresponding descriptor for the predicate (AGE = 42). Let us say that the descriptor in DDIT happens to be (41 =< AGE =< 45). We then have to find all the descriptors whose attribute values are greater than the attribute values of this descriptor. The sequence set of a B+tree makes it very simple and efficient to find all the descriptor values greater than (or less than) a given descriptor value.

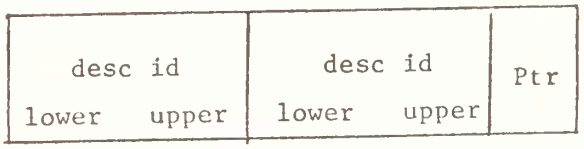
A sample B+tree containing the descriptors for a directory attribute is shown in Figure 23. In this example, there are at most three attribute values in each node of the index part of the tree and at most two descriptors in each node of the sequence-set part of the tree. Along with a descriptor in the sequence set, there is a pointer to the first set of bits in the bit map for the descriptor. (The bit map for a descriptor is used in the cluster search phase of directory management and is described in the next section.) Let us illustrate the descriptor search by means of an example. Consider the DDIT for the directory attribute AGE depicted in Figure 24. (The pointer to the bit maps are not shown in the figure because they are not relevant to our



A. The B+tree



B. The Format of a Node in the Index part of the tree



C. The Format of a Node in the Sequence Set Part of the Tree

Figure 23. A Sample B+tree Containing the Descriptors for a Directory Attribute

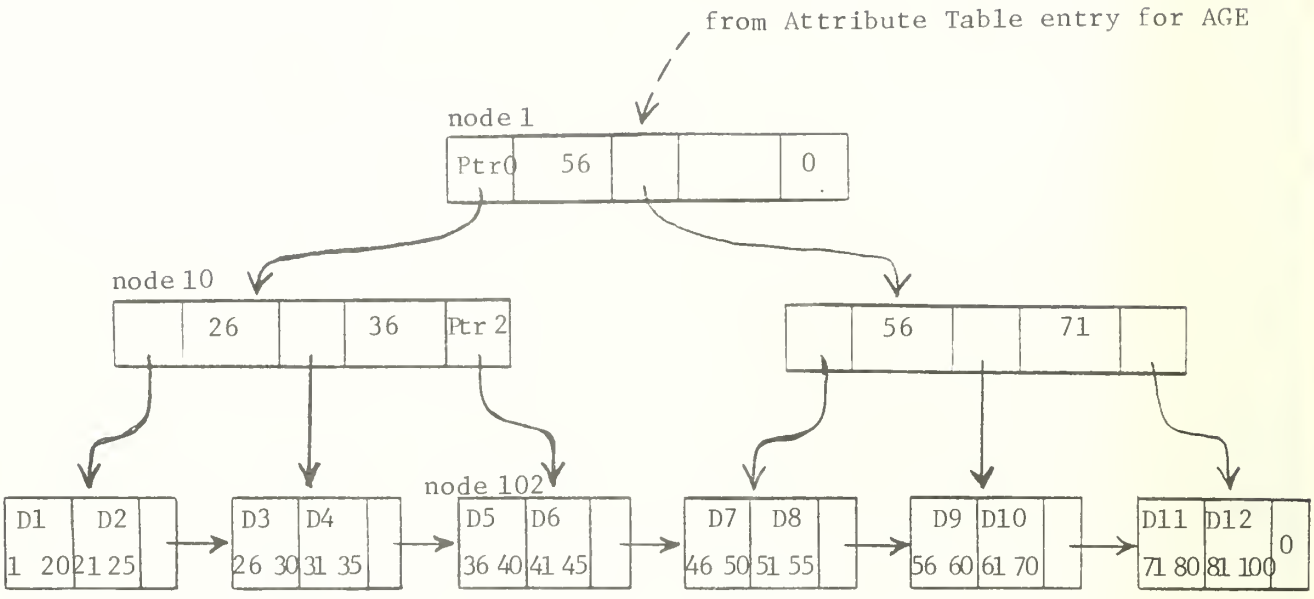


Figure 24. An Example of the Descriptor Search Phase

discussion here.) The pointer to the root of this B+tree is obtained from the attribute table. Let us assume that we want to find the corresponding set of descriptor ids for the predicate (AGE >= 42). We have to first find the corresponding descriptor id for the predicate (AGE = 42). The root node (node1) is searched for the value 42. Since 42 < 56, the subtree pointed to by ptr0 (in node1) should be searched next. Since 42 > 36, the subtree pointed to by ptr2 (in node10) should be searched next. We note that node102 is at the sequence-set level. Since 41 =< 42 =< 45, the descriptor id D6 is the corresponding descriptor id for the predicate (AGE = 42). We now have to find all the descriptors whose attribute values are greater than the values of the descriptor 41 =< AGE =< 45, since the predicate is (AGE >= 42). This is done by following the pointer associated with node102 and taking all the descriptors. Thus, the corresponding set of descriptor ids for the predicate (AGE >= 42) is {D6, D7, D8, D9, D10, D11, D12}.

4.3. The Cluster-Definition Table (CDT)

We recall that this table is used in the cluster search and address generation phases of directory management. In the cluster search phase, the clusters whose records may satisfy a request are determined. In the address generation phase, the secondary storage addresses of the clusters are determined. A sample CDT is depicted in Figure 25.

Two tables are used to implement the cluster-definition table. The first table, descriptor-id-cluster-id-bit-map table (DCBMT), represents a two way mapping (see Figure 26). It shows, looking at a column of the table, which descriptor ids are in the descriptor-id set for a cluster. For example, the descriptor ids D1, D2, D3, D5, D7 and D8 are in the descriptor-id set for cluster C4. The table also shows, looking at a row of the table, which clusters have a descriptor id in their descriptor-id sets. For example, the clusters C1, C3, C4, and C8 have the descriptor id D2 in their descriptor-id sets. The DCBMT is used to perform the cluster search. The second table, cluster-id-to-secondary-storage-address table (CSSAT), contains the secondary-storage addresses for clusters (see Figure 27). In this table, space is reserved for two addresses (for each cluster) and the extra addresses are represented using links. In the following sections, we describe the two tables in detail.

Cluster Number	Corresponding Set of Descriptor Ids	Address of the Record in the Cluster
C1	D2,D3	A1234,A1235
C2	D1,D4	A3456,A3457
C3	D2,D3,D4	A5676
	.	
	.	
	.	

Figure 25. A Sample of The Cluster-Definition Table (CDT)

Cluster id / descriptor id	C1	C2	C3	C4	C5	C6	C7	C8	. . .
D1	0	1	0	1	1	0	0	1	. . .
D2	1	0	1	1	0	0	0	1	. . .
D3	1	0	1	1	0	1	0	1	. . .
D4	0	1	0	0	1	1	1	0	. . .
D5	0	0	1	1	0	1	0	1	. . .
D6	1	1	0	0	1	1	0	1	. . .
D7	1	0	1	1	1	0	1	1	. . .
D8	1	0	0	1	0	1	1	0	
.	.	.	.	.	.	.	.	.	
.	.	.	.	.	.	.	.	.	
.	.	.	.	.	.	.	.	.	

Fiture 26. A Sample of the Descriptor-Id-Cluster-Id-Bit-Map Table (DCBMT)

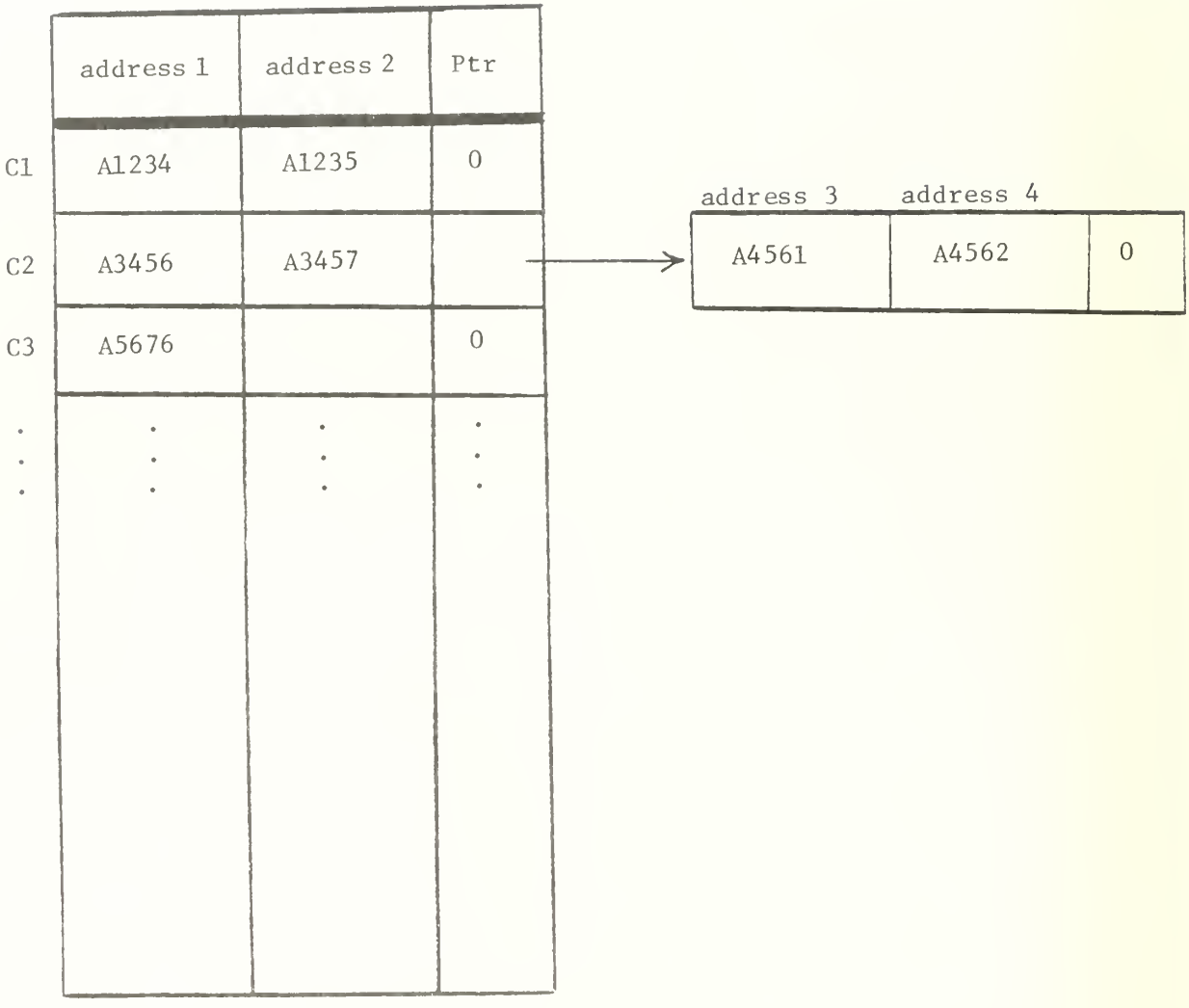


Figure 27. A Sample of The
Cluster-Id-to-Secondary-Storage-Address Table (CSSAT)

4.3.1. The Descriptor-Id-and-Cluster-Id-Bit-Map Table (DCBMT)

In the implementation of DCBMT, the table can be organized to allow efficient access to only columns, only rows or both columns and rows. We recall that the input to the cluster search phase is a descriptor-id group. Thus, the DCBMT should be organized around the descriptor ids, i.e., it should allow efficient access to the rows of the table. However, the bit map for a descriptor id, a row of the table, grows as new records are inserted and new clusters are generated. Thus, each row of the DCBMT is organized as a linked list. A sample of the resulting DCBMT is depicted in Figure 28. The pointers to the first set of bits in the bit map for descriptor ids are obtained from the DDIT. Along with the first set of bits in the bit map for a descriptor id, there is a pointer to the next set of bits in the bit map for the descriptor id. Let us illustrate the cluster search by means of an example. Consider the DCBMT depicted in Figure 29, and let us assume that we are searching for clusters whose descriptor-id sets contain the descriptor-id group {D1, D3, D7}. We need to find the bit map for each descriptor id in the descriptor-id group and then AND (logical and) the bit maps. From the DDIT, the pointer to the first set of bits in the bit map for D1 is found. Using this pointer, the first set of bits (0101) in the bit map is obtained. Using the pointer associated with the first set of bits, the second set of bits (1001) in the bit map for D1 is obtained. Since the pointer associated with the second set of bits is null, there are no more bits in the map. So, the bit map for D1 is 01011001. In a similar way, the bit maps for D3 and D7 are obtained. Let us assume that they are 10110101 and 10111011, respectively. Now, the three bit maps are ANDed:

	C1	C2	C3	C4	C5	C6	C7	C8
D1	0	1	0	1	1	0	0	1
D3	1	0	1	1	0	1	0	1
D7	1	0	1	1	1	0	1	1
	0	0	0	1	0	0	0	1

Thus, the descriptor-id sets for the clusters C4 and C8 contain the descriptor-id group {D1, D3, D7}.

4.3.2. The Cluster-Id-To-Secondary-Storage-Address Table (CSSAT)

After determination of the clusters, the secondary-storage addresses are obtained from the CSSAT. The number of addresses for a cluster grows as new

Figure 28. A Sample DCBMT Stored as an Indexed-Linked List

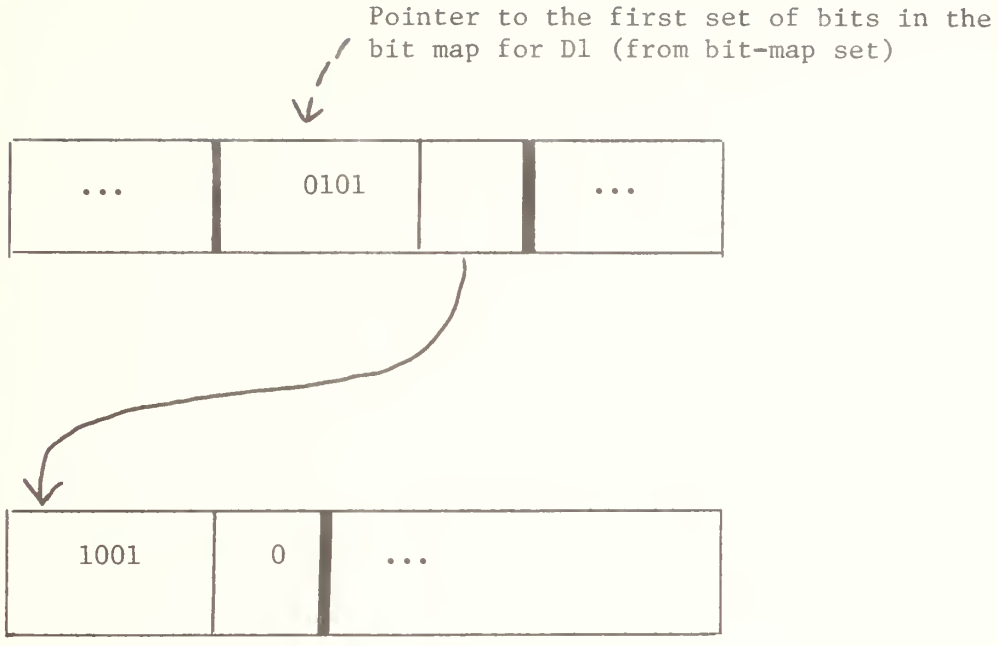


Figure 29. An Example of the Cluster Search Phase

records (belonging to the same cluster) are inserted. Thus, the addresses for a cluster are kept as a linked list. A sample of the resulting CSSAT is depicted in Figure 30. A "cluster-address index" is used to obtain the first set of secondary-storage addresses for clusters. Along with the first set of addresses for a cluster, there is a pointer to the next set of addresses for the cluster. The cluster-address index is very small in size. Let us illustrate this by means of an example. We note that there are as many entries in the cluster-address index as there are cluster-address sets. So, in order to find the number of entries in the cluster-address index, we need to find the number of cluster-address sets. Let us assume

```
track size = 25K bytes
number of tracks in a backend = 20K
cluster size = 1 track
one cluster-address set size = 1 track
one secondary-storage address (in cluster-address
                                        set) = 6 bytes
one pointer to the next set of addresses (in
                        cluster-address set) = 8 bytes
```

In order to find the number of cluster-address sets, we need to find how many clusters fit in one cluster-address set. Using the above data we calculate

```
space used by one cluster in cluster-address set =
   two secondary-storage addresses + pointer to the
      next set of addresses = (2 * 6 + 8 = 20) bytes
number of clusters in one cluster-address set =
                                    25K / 20 = 1.25K
number of clusters in a backend = 20K
number of cluster-address sets = 20K / 1.25K = 16
```

So, there are only 16 entries in the cluster-address index for 20K tracks. Thus, this table can be kept in the primary memory. (If there are more tracks, say 100K tracks, the number of entries in the cluster-address index is still small, only 5 * 16 = 80 entries.

Let us illustrate the address generation by means of an example. Consider the CSSAT depicted in Figure 31, and let us assume that we want to find the addresses for cluster 2 (C2). Using the cluster-address index, the first set of addresses (A3456, A3457) for C2 is obtained. Using the pointer associated with the first set of addresses, the second set of addresses (A4561, A4562) for C2 is obtained. Since the pointer associated with the second set of addresses is null, there are no more addresses for C2. Thus, the addresses for C2 are (A3456, A3457, A4561, A4562).

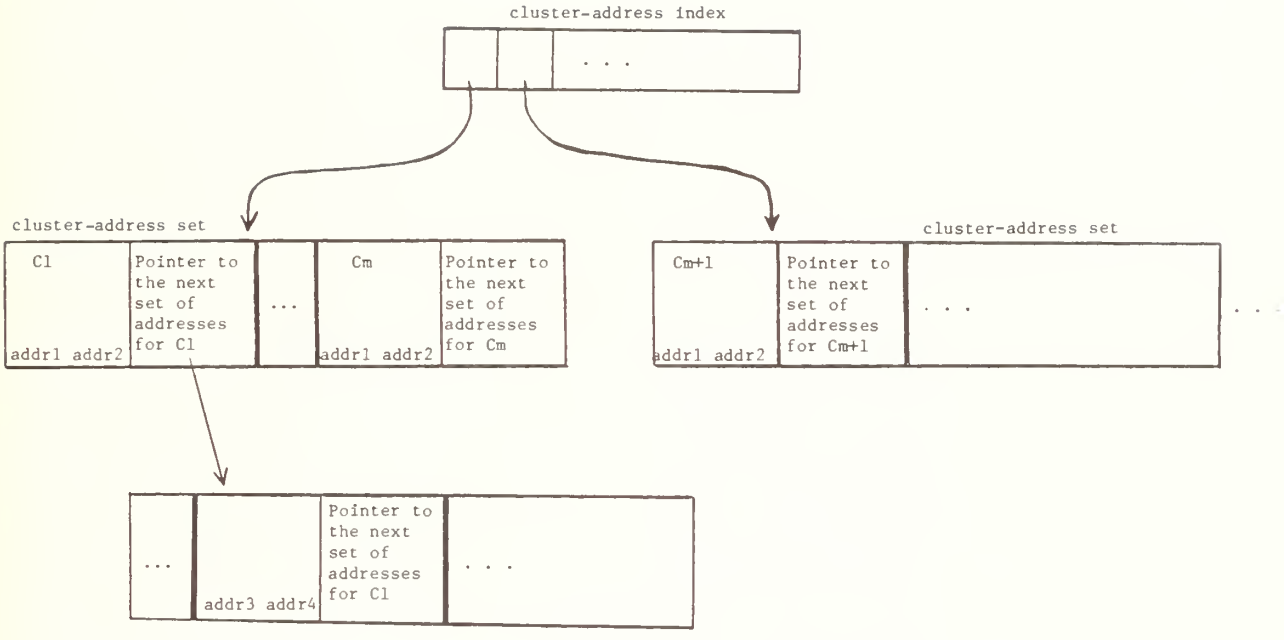


Figure 30. A Sample CSSAT Stored as an Indexed-Linked List

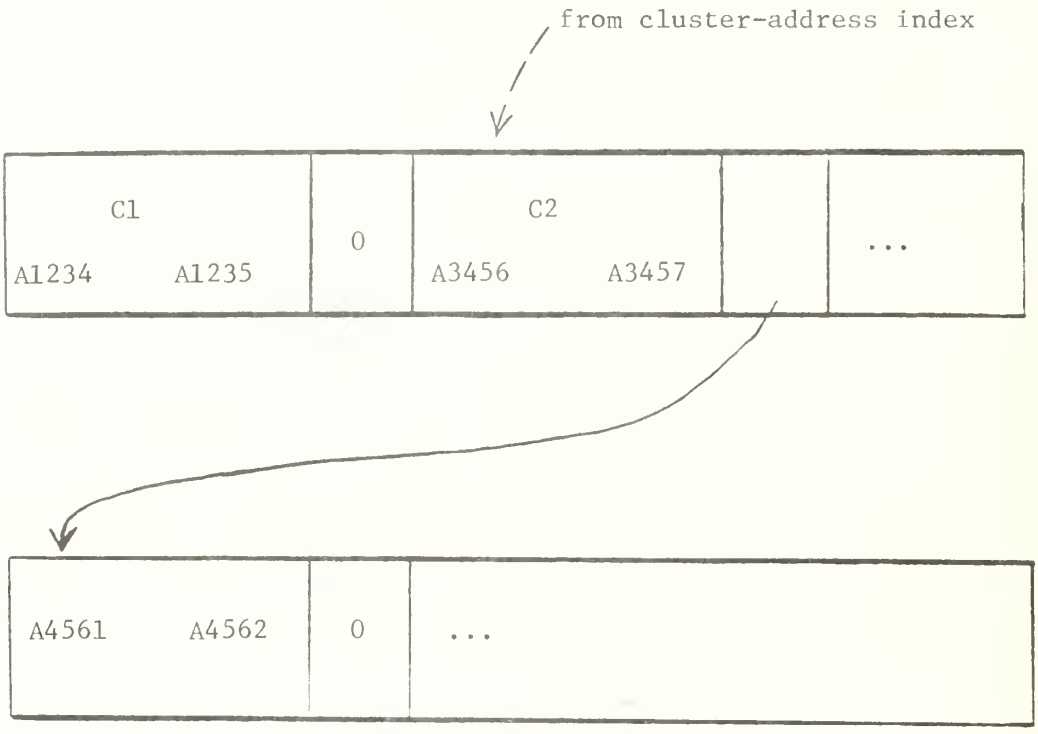


Figure 31. An Example of the Address Generation Phase

5. OUR IMPLEMENTATION EXPERIENCE

As described before, multiple versions of MDBS are being developed. Some versions have been completely developed and tested. In this chapter, we first discuss some system issues that arose during the development. We then describe our software-engineering observations, mostly during the integration of different components.

5.1. System Issues

Since most of MDBS is completed at this time it is possible to describe the size of the actual system. Recall that MDBS consists of five processes running in the controller (request preparation, insert information generation, post processing, get pcl and put pcl) as well as another five processes running on each of the backends (directory management, record processing, concurrency control, get pcl and put pcl).

5.1.1. The Controller

MDBS contains five processes in the controller, request preparation, insert information generation, post processing, get pcl and put pcl. The C source programs for these processes are about 3600 lines of which 50% are comments, leaving 1800 lines of actual code. The space required for the object code is as follows:

Request preparation	50K
Insert information generation	28K
Post processing	24K
Get pcl	23K
Put pcl	23K
Total	148K

It can thus be seen that the largest part of the controller is request preparation. Its size is due to the parser which is its main component.

5.1.2. The Backends

The C source programs for each backend is about 7600 lines of which 45% are comments leaving about 4200 lines of actual code. This code is divided among 165 procedures. The space required for object code in each backend is shown below:

Primary-memory-based directory management	58KB
Record processing	50KB
Concurrency control	43KB
Get pcl	23KB
Put pcl	23KB
Total	197KB

The operating system itself requires 74K bytes. Additional space is required for the messages that are being passed between the processes. Finally, each buffer for disk input and output requires 16K bytes. Thus, if there are to be four buffers, say, for disk input and output, more than 335K (197 + 74 + 4*16) bytes of physical memory are required for each MDBS backend. As will be seen shortly, the required physical memory can be reduced significantly by the use of overlays.

Because of their 16-bit addresses, PDP-11s allow only a 64K-byte virtual address space for each process. RSX requires 8K bytes of virtual address space for the messages being used by a process. Thus there is only 56K bytes of virtual address space available for the rest of each process. Thus directory management must use an overlay strategy to have enough virtual space. Because record processing requires access to the large input/output buffers, it must also be overlaid. Fortunately, it was possible to overlay the process initialization routines. These routines, in fact require 8K bytes. Thus once MDBS is initialized, no additional overlay processing is required.

The space shown above actually includes statements that print intermediate results while MDBS is executing. In fact about 6K bytes are required in each process if we are going to allow printing of intermediate results. Such output, of course, is not required in a production system. However, output during system integration and testing was almost certainly required.

We have not yet converted all the processes to non-printing versions. However, we can make the following estimates for the size of these versions.

Primary-memory-based directory management	44KB
Record processing	36KB
Concurrency control	29KB
Get pcl	9KB
Put pcl	9KB
Total	127KB

Thus a production system on a machine with 256K bytes of physical memory would

have space available for several input/output buffers as well as for the messages being passed between processes. In addition, we note that each process listed above requires a virtual address space smaller than the maximum virtual address space (i.e., 64K bytes) provided by the PDP-11.

5.1.3. The Test Interface

We have chosen to implement our test interface in the controller also. This interface allows us to create and modify files of test requests. These requests can then be submitted to MDBS by giving a single command. This interface makes it easy to run demonstrations and tests of MDBS.

The test interface consists of about 1700 C statements of which about 55% or 950 statements are executable. The object code for the test interface requires 39K bytes.

5.1.4. The Disk Input/Output Interface

In the preceding sections we discussed the size of the MDBS system. In this section we discuss one of the major problems we had during implementation. This problem occurred with the low-level interface required to actually read and write a track of data from or to the disk. Because MDBS is planned to be an efficient database system, it is necessary to choose the right interface for actual input and output from and to the disk. Many alternatives were available to us. We could use a very low-level interface with the disk device driver software. We could also use a very high-level file system interface. Our goal was to minimize our work while still obtaining good system performance. Thus we decided that neither of these approaches was appropriate, one was at too low a level, the other at too high a level. Thus we looked for a level in between. Because of our lack of experience with the RSX operating system and because of poor documentation of what exactly was available, this interface was very hard for us to develop.

The disk drives we are using on our PDP-11/44's perform actual input and output on 512-byte blocks. Below the file level, RSX provides three types of input/output support. These are called virtual, logical and physical I/O. The physical I/O interface requires the user to specify the actual physical block address on the disk. In this case, the user must also keep track of any bad blocks on the disk. At the logical I/O interface, the system automatically

skips any bad disk blocks. Thus the user does not have to manage these bad physical blocks. At the virtual I/O level, the user is allowed to use arbitrary record size rather than being restricted to 512 byte blocks.

Our major difficulty was in determining what these actual interfaces were since most documentation only discusses the file system interface. Once we finally understood these three interfaces, it was not hard to choose the logical I/O interface. We rejected the virtual I/O level because we were actually trying to read or write a whole track of data (which is, of course, of fixed size) and because we wanted to make our system as efficient as possible. We decided to use the logical rather than physical level since we would then not be required to develop software to manage any bad blocks on the disks. Since the purpose of our implementation of MDBS is to show its feasibility, our experiments will use only disks without bad blocks of data. Thus our performance will be as good as if we had used physical I/O. However we would be relieved of having to handle any bad blocks ourselves.

5.2. Our Software-Engineering Observations

The software engineering techniques that we have employed are described in [Kerr82]. The effectiveness of the techniques is described in [He82]. Therefore, we will not repeat them here. We will, however, describe some problems that we have observed during the development, integration and testing of MDBS. We should point out that most of our observations have been stated by other software engineers. So, we are mainly confirming the problems rather than defining new problems.

5.2.1. Use of Standards in Coding

We did not set any standards to be used by all programmers. So, every programmer, when coding, followed his style for choosing variable/subroutine names and indenting the program. For example, the following are three ways that the if-then-else construct was indented by different programmers.

```
1)
  if ( logical expression )
     {
     statements
     }
  else
     {
     statements
     }
```

```
2)
  if ( logical expression ) {
     statements
  }
  else {
     statements
  }


3)
  if ( logical expression )
  {
     statements
  }
  else
  {
     statements
  }
```

Lack of standards in coding caused us some problems. First, the code written by a project member is not easily comprehensible by the other project members. Second, when a programmer is assigned to continue the coding done by a previous programmer, he may decide that it is hard for him to read the code. Thus, he would change the code to his own style.

A second problem occurred when we combined different programs written by different programmers. We ran into the problem of multiply defined names. Because we did not maintain a dictionary of program names, we discovered that several names had multiple definitions.

To overcome these problems, standards should be set for choosing variable/subroutine names and indenting the program. These standards must then be followed by all the project members.

5.2.2. The System Programming Language and the Language Compiler

As described in Chapter 1, PDP-11/44's are used for the MDBS backends. The operating system used is RSX-11M, the programming language used is C and the compiler used is DECUS C. Some minor problems were caused by the language and the compiler. First, the compiler requires that the variable names be different in their first eight characters and the function names be different in their first six characters. The truncation to six (or eight) characters caused additional multiply defined names when we integrated different parts written by different programmers.

Second, the C language does not include the concept of internal procedures. Use of internal procedures give a better structure to the system, especially when implementing data abstractions. It also avoids some of the multiply defined names when integrating different components. For example, consider the following two PL/I-like program segments.

```
1)
PROC Q;
   ...
   PROC ADD;
   ...
   END ADD;
END Q;
```

```
2)
PROC R;
   ...
   PROC ADD;
   ...
   END ADD;
END R;
```

A block-structured language allows these two segments to be added to a program without creating any multiply defined names. In C, the two program segments would be

```
1)
q()
{...
}
add()
{
}...
```

```
2)
r()
{
}...
add()
{
}...
```

Adding these two segments to a program creates a multiply defined name, specifically, "add".

Another problem caused by the language is that integer variables and pointers are passed between routines as "call by value" in C. In order to get

values for some variables back from a routine, one can use the "return value" of the function for one variable, and must pass pointers to the other variables. Passing pointers to pointers, though it works fine, does not provide clear and easily comprehensible programs. Let us illustrate this point by briefly describing the parser function in the request-preparation process (in the controller). The input to the parser is a traffic unit, i.e., either a request or a transaction. If there are no syntax errors in the traffic unit, the parser allocates space for and stores the parsed-traffic unit. It also allocates space for and stores the aggregate operators. It then returns a pointer to the parsed-traffic unit as well as a pointer to the aggregate operators. The data structures used in the request-preparation process is shown below.

```
struct  req_definition  {
                            ...
                        };
struct  agg_definition  {
                            ...
                        };
```

The main routine of the request-preparation process, shown below, will call the parser. The main routine expects the parser to return two pointers.

```
main()  /* Request-Preparation */
{
   struct  req_definition  *ParsedTrafUnitPtr,
                           *parser();
   struct  agg_definition  *AggOpsPtr;
   ...
   ParsedTrafUnitPtr = parser(&AggOpsPtr, ...);
   ...
}
```

The parser, shown below, will return two pointers to the main routine of the request-preparation process.

```
struct  req_definition  *parser(p, ...)
   int     *p;
   ...
{
   struct  req_definition  *ParsedTrafUnitPtr;
   ...
   ParsedTrafUnitPtr = ...;
   *p = ...;
   ...
   return(ParsedTrafUnitPtr);
}
```

One pointer, the pointer to the parsed-traffic unit, is returned using the

"return value". But, the other pointer, the pointer to the aggregate operators, is returned by accepting a pointer to the pointer.

Finally, to produce efficient code, uninitialized variable reference and out of bound array reference are not checked. Obviously, the final system program should be as efficient as possible, so these checks could be excluded. However, it is helpful to have these facilities during the testing. That is, a compiler that produces codes for the checking would be helpful. This compiler would be used while testing the system. When the system is completely tested, the compiler that produces efficient code, by excluding the checking, can then be used to produce the final version of the system.

5.2.3. High-Level Design vs. Low-Level Design

The design of a system specifies "what" the system does. It does not specify "how" to implement the system. The system specification language (SSL) employed in the MDBS project allows for both high-level design and low-level (detailed) design. The critical routines are specified in more detail by the designers. When coding the routines specified in more detail, the programmers usually translate the design directly to the system programming language used, namely, C. Thus, the special cases, such as checks for not exceeding array sizes, that are not usually included in the design are overlooked by the programmers and cause errors.

5.2.4. The Importance of Interfaces Between Processes

When a system consists of more than one process, the interfaces between the processes should be specified before developing each process in detail. This is because the change in one process could affect another process if there is no specified interface between the two processes.

As described in Chapter 2, there are multiple processes in MDBS. We did not completely specify the interfaces between processes in the controller and backends. This caused some changes to some processes after they had been completely developed. For example, after the insert-information-generation process in the controller was developed, we had to modify it somewhat to provide some information required in the record-processing process in the backends. (The information required was an indication of whether or not a record being inserted is the first record on a track.) As another example, we started cod-

ing the directory-management process in the backends before defining completely the process structure and interfaces between processes in MDBS. As a result, we had to modify the directory-management process after we designed the process structure and interfaces between processes in MDBS.

5.2.5. The Problems in a University Environment

The MDBS-project members are faculty and students. The project members have other responsibilities, such as teaching and/or taking courses. Thus, this is not a group of people working solely on the project. This has certainly slowed down the development.

Initially, we had no members with knowledge of the underlying operating systems. So, we had to learn all the details of the operating systems that we needed. This has taken a major portion of the development time.

Project members leave as they graduate, and new members join the project. (Since the start of the project, four members have graduated and seven other members have left the project.) It takes about three months to train a new member, since he has to learn the MDBS design, the system specification language (SSL) used, the system programming language, and/or some details of the underlying operating systems. This has also slowed down the development.

REFERENCES

[Come79] Comer, D., "The Ubiquitous B-Tree," Computing Surveys, Vol. 11, No. 2, June 1979, pp. 121-137.

[DEC79a] "PCL11-B Parallel Communication Link Differential TDM Bus," Digital Equipment Corp., Maynard, Mass., 1979.

[DEC79b] "RSX-11M/M-PLUS Executive Reference Manual," AA-H265A-TC, Digital Equipment Corp., Maynard, Mass., 1979.

[DEC80] "VAX/VMS System Services Reference Manual," AA-D018B-TE, Digital Equipment Corp., Maynard, Mass., 1980.

[He82] He, X., et al., "The Implementation of a Multi-Backend Database System (MDBS): Part II - The First Prototype MDBS and the Software Engineering Experience," Technical Report, NPS-52-82-008, Naval Postgraduate School, Monterey, California, July 1982.

[Hsia81a] Hsiao, D.K. and Menon, M.J., "Design and Analysis of a Multi-Backend Database System for Performance Improvement, Functionality Expansion and Capacity Growth (Part I)," Technical Report, OSU-CISRC-TR-81-7, The Ohio State University, Columbus, Ohio, July 1981.

[Hsia81b] Hsiao, D.K. and Menon, M.J., "Design and Analysis of a Multi-Backend Database System for performance Improvement, Functionality Expansion and Capacity Growth (Part II)," Technical Report, OSU-CISRC-TR-81-8, The Ohio State University, Columbus, Ohio, August 1981.

[Kerr82] Kerr, D.S., et al., "The Implementation of a Multi-Backend Database System (MDBS): Part I - Software Engineering Strategies and Efforts Towards a Prototype MDBS," Technical Report, OSU-CISRC-TR-82-1, The Ohio State University, Columbus, Ohio, January 1982.

APPENDIX A
HOW TO READ AND FOLLOW THE PROGRAM SPECIFICATIONS

The appendices in this series have contained the detailed design of MDBS. In Appendix B, the programs for the concurrency control process are described and specified. These programs represent those parts of MDBS that have been designed, since the first two reports in this series were written.

A.1 Parts within an Appendix

Each appendix begins with a introduction which outlines the major components of the design. For example, the design of the controller subsystem, presented in [He82], consisted of three major parts: request preparation, insert information generation and post processing. The design of a backend subsystem also consists of three major parts: directory management, record processing and concurrency control. The first two parts, directory management and record processing, were presented in the previous reports. The last part, namely, concurrency control, in presented in Appendix B of this report.

A.2 The Format of a Part

In each part, we provide the following documentation elements:

(1) Title of the part,
(2) Name of the design,
(3) Name of the designer,
(4) Date the design was first submitted,
(5) Dates of design modifications,
(6) Statements of the design purpose, and of the input and output requirements,
(7) Formal specifications of the input and output, if necessary,
(8) Procedure names used in the design,
(9) Jackson chart of the design, if necessary,
(10) Data structures used in the design,
(11) Program specification of the design.

A.3 Documentation Techniques for a Part

In the previous section, we listed the various documentation elements. They are used to describe a design. Documentation elements 1 through 5 are written in English phrases. Document element 6 is written in prose. On the other hand, document elements 7 through 11 can be expressed more effectively using other means. Specifically, we have used Backus-Naur form (BNF) for writing the specifications in document element 7.

The procedure names of document element 8 are shown in a program hierarchy. The use of the hierarchy makes clear the calling sequences of the procedures named. The data structures of documentation element 10 are specified in either the system specification language (SSL) or in the C programming language. In documentation element 11, the procedures, themselves, are specified in SSL.

Except for the programming team that writes the procedures, other teams will usually not be interested in the internal logic of the procedures. Consequently, they need only know the higher-level specifications of the procedures. The SSL employed in MDBS is an ideal specification language for revealing the design of the procedures from a top-to-bottom-and-layer-to-layer way. It also works well with the hierarchical organization of procedures.

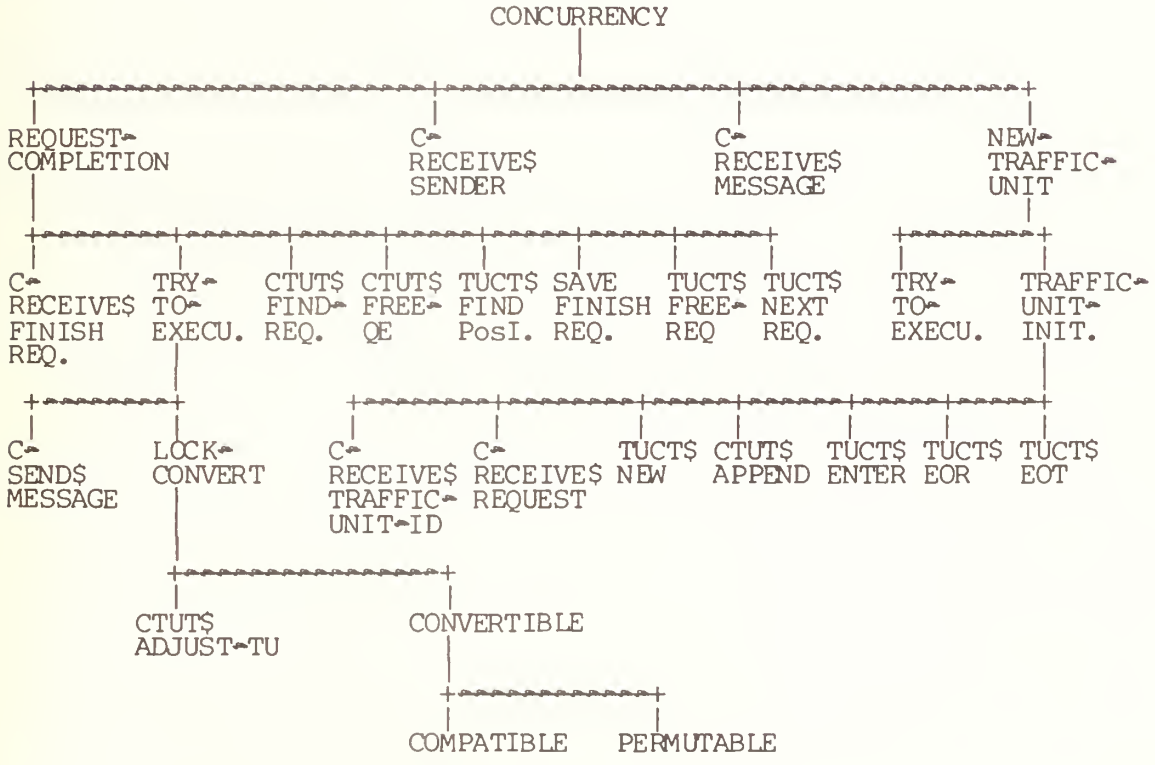
APPENDIX B

THE SSL SPECIFICATIONS FOR MDBS CONCURRENCY CONTROL

The system specification for the message-oriented concurrency control process is given in this appendix.

```
/*    (1) Message-Oriented Concurrency Control                    */
/*    (2) Design      : CONCURRENCY                               */
/*    (3) Designers   : Steven A. Demurjian, Douglas S. Kerr,     */
/*                      and Patti Dock.                           */
/*    (4) Date        : November 29, 1982                         */
/*                      December 2,  1982                         */
/*    (5) Modified    : January 13, 1983                          */
/*                      February 1, 1983                          */
/*    (6) Purpose     :                                           */
/*        This is the Concurrency Control Process.  The goal of   */
/* concurrency control is to insure the consistency of the       */
/* database, while allowing concurrent execution of multiple     */
/* requests.  The concurrency control process consists of        */
/* the functions that must be performed when Directory Man-       */
/* agement signals there is a new traffic unit, or Record Pro-   */
/* cessing signals that a request has finished execution.        */
```

(8) Procedure Hierarchy for CONCURRENCY



(10) Data Structures

```
/*     The data structures definitions are included in the program */
/* specifications.                                                */
```

(11) Program Specifications

```
1.  task CONCURRENCY

2.      do initialization work;
3.      while 'true' do /* do forever */
4.         if according to the task scheduling algorithm this task should
                                                    release the processor
5.            then
6.               release the processor and wait;
7.         end if;
           /* Get the next message for CONCURRENCY */
8.         perform C-RECEIVE$MESSAGE;
           /* Get the sender of the next message for CONCURRENCY.  */
9.         perform C-RECEIVE$SENDER(sender);
10.        case  sender   value
11.              'DIRECTORY-MANAGEMENT':
                    /* A new traffic unit is being received. */
12.                 perform NEW-TRAFFIC-UNIT;
13.              'RECORD-PROCESSING':
                    /* A request has completed execution. */
14.                 perform REQUEST-COMPLETION;
15.              'otherwise':
16.                 system error;
17.        end case;
18.     end while;
19. end task;

12.1  procedure NEW-TRAFFIC-UNIT;
      /* A new traffic-unit is to be received.  We must make entries in
         the Traffic-Unit-to-Cluster-Table (TUCT) and the Cluster-to-
         Traffic-Unit-Table (CTUT), marking all requests as "to-be-used".
         We must also try to execute the first request in the traffic unit. */

      /*  TU is the traffic unit id of the traffic unit just received.
          Req is the number of the first request in TU.  PositionInReq is
          the position of the first entry in TUCT for the request Req.
          All three of these values are returned from
          TRAFFIC-UNIT-INITIALIZATION.   */
12.2      perform TRAFFIC-UNIT-INITIALIZATION( TU , Req , PositionInReq);

      /*  Try to execute the first request in the traffic unit, i.e. Req,
          starting at the position PositionInReq.  */
12.3      perform TRY-TO-EXECUTE( TU , Req , PositionInReq );

12.4    end procedure;
```

```
12.2.1  procedure TRAFFIC-UNIT-INITIALIZATION( Output: TU , Req , PosInReq );

            /* Setup TUCT and CTUT entries for the new traffic unit.
               TU is the traffic unit id  Req is the request number for
               the first request in the traffic unit TU, and PosInReq is the
               first position within the request Req.  */

            /* Get the traffic unit id. */
12.2.2          perform C-RECEIVE$TRAFFIC-UNIT-ID( TrafficUnitId );

            /* Get an entry for the queue element which indexes the requests
               for the traffic unit TU.  The queue element has the form:
               { TU   : The traffic unit id;
                 #REQ : The number of requests in the traffic unit;  }  */
12.2.3      perform  TUCT$NEW( TrafficUnitId , TU );
            /* Enter all the requests for this traffic unit into TUCT and
               CTUT.     */
             /* Get the first request and corresponding list of clusters. */
12.2.4          perform C-RECEIVE$REQUEST( Request );
12.2.5          Req = Request;  /*Set Req to the first request in TU */
             /* Enter the requests. */
12.2.6         while there are more requests in this traffic unit do
                 /* Enter the request in the TUCT and in the CTUT as
                    appropriate. */
12.2.7           For the TUCT table, build a queue element TUCT-QE
                 of the form :
12.2.8           {  Req = The request number for a request in the
                           traffic unit;
12.2.9              ClusterId = The cluster identification number;
12.2.10             Mode = mode of the request;
12.2.11             Category = "to-be-used";   }

12.2.12          For the CTUT table, build a queue element CTUT-QE
                 of the form :
12.2.13          {  TU = The traffic unit id;
12.2.14             Req = The request number;
12.2.15             Mode = mode of the request;
12.2.16             Category = "to-be-used";   }
12.2.17       TU.#REQ = TU.#REQ + 1 ;
12.2.18       for each cluster, ClusterId, needed by the request do

                 /* Enter QE into the queue for cluster ClusterId. */
12.2.19             perform  CTUT$APPEND( ClusterId , CTUT-QE );

                 /* Enter QE into TUCT. */
12.2.20             perform  TUCT$ENTER( TU , TUCT-QE );
12.2.21       end for;

              /*Enter end-of-request(EOR) into TUCT. */
12.2.22       perform TUCT$EOR;
              /* Get the next request and list of clusters.*/
12.2.23       perform C-RECEIVE$REQUEST(Request);
12.2.24     end while;
          /*Enter end-of-transaction(EOT) into TUCT. */
12.2.25   perform TUCT$EOT;
12.2.26 end procedure;
```

```
14.1  procedure REQUEST-COMPLETION;
       /* A request has finished execution. That request must be removed
          from the Cluster-to-Traffic-Unit-Table.  Any requests that were
          deactivated by this request must be reactivated.  For each request
          that was reactivated, we must try to execute that request.  The
          next request in this traffic unit must also be activated.   */

          /* get the request */
14.2      perform C-RECEIVE$FINISHED-REQUEST( FinishedRequestId );

          /* Find the Finished-Request in TUCT. */
          /* TU is the traffic unit id and Req is the request number
             in the traffic unit TU that just finished.  PosInReq is the
             first position in the queue for the request Req and the traffic
             unit TU.  */
14.3      perform TUCT$FIND( FinishedRequestId , TU , Req , PosInReq );

          /* Save this id for later use. This procedure is not being
             utilized at the present time.  */
14.4      perform SAVE$FINISHED-REQUEST( FinishedRequestId );
14.5      ReqPos = PosInReq;

          /* Pictorial description of the current status of the TUCT queue
             for the traffic unit TU and the request Req that just finished.


             TU --> Earlier          EOR  Req          EOR  Later   EOR EOT
                    Requests(if any)     ClusterId's       Requests(if any)
                                          |
                                         PosInReq
                                         ReqPos

             PosInReq and ReqPos both initially reference the first queue
             element which contains the first ClusterId for the request Req.
             The outside while loop operates as long as there are still
             ClusterId's to examine for the request Req.          */

14.6      while  ( ReqPos <> EOR ) do
            /* Find the position in the CTUT table where the pair (TU , Req)
               lives.
               The cluster to look at is ReqPos.ClusterId.  Put the result in
               PosInClusQ(Position In Cluster Queue).  */
14.7        perform  CTUT$FIND-REQUEST( ReqPos.ClusterId , TU , Req ,
                                                          PosInClusQ );

            /* Pictorial description of the cluster ReqPos.ClusterId in
               the CTUT table that contains the pair (TU , Req).

               ReqPos.ClusterId --> .....  TU  Req  ..... EOQ
                                           Mode
                                           Category

               The "...." represents other queue elements that need the
               cluster ReqPos.ClusterId.

               At this point, the CTUT queue element for the pair (TU , Req)
               is removed from the queue.  Then a pointer, ClusPos, is set
               to the first position in the cluster queue ReqPos.ClusterId.
               The while loop below will then cycle through all elements in
               the queue ReqPos.ClusterId to check if any other (traffic unit,
               request) pair, specifically (ClusPos.TU , ClusPos.Req) was
               "waiting" for the pair (TU , Req) that just finished.  */

14.8        perform  CTUT$FREE-QE( PosInClusQ );
14.9        ClusPos = "First position in cluster queue ReqPos.ClusterId"
```

```
14.10       while ( ClusPos <> "null" ) do
14.11          if ClusPos.category = "waiting"
14.12             then
                   /* The traffic unit which is "waiting" is ClusPos.TU.
                      The request within that traffic unit is ClusPos.Req.
                      Now find the position of the cluster ReqPos.ClusterId
                      in the TUCT queue for the pair (ClusPos.TU ,
                      ClusPos.Req).  The position within the TUCT queue is
                      returned in the pointer PosInReqB.  A picture of the
                      TUCT queue appears below:

                      ClusPos.TU : Earlier EOR ClusPos.Req EOR Later EOR EOT
                                   Requests    ClusterId's     Requests
                                               |
                                               PosInReqB
                                                                        */

14.13                 perform TUCT$FINDPosInReq( ClusPos.TU , ClusPos.Req ,
                                                 ReqPos.ClusterId , PosInReqB );

                      /* Now try to execute the Request starting at PosInReqB.
                                                                        */
14.14                 perform TRY-TO-EXECUTE( ClusPos.TU , ClusPos.Req ,
                                                           PosInReqB );
14.15              end if;
14.16              ClusPos = ClusPos + 1;
14.17          end while;
14.18          ReqPos = ReqPos + 1;
14.19       end while;

            /*Remove the request Req from the traffic unit TU */
14.20       perform  TUCT$FREE-REQ( PosInReq );

14.21       if "queue TU is empty"
14.22          then
                 /* Remove the traffic unit TU from TUCT */
14.23          else
                 /* Get the next request in the traffic unit TU and try to
                    execute the request. */
14.24            perform TUCT$NEXT-REQUEST( TU , Req , PosInReqC );
14.25            perform TRY-TO-EXECUTE( TU , Req , PosInReqC );
14.26       end if;
14.27    end procedure;


12.3.1   procedure TRY-TO-EXECUTE( TU , Req , PosInReq );


         /* TU is a traffic unit from the TUCT table.  PosInReq is the position
            in the TU (the specific cluster for a Req) where the procedure
            LOCK-CONVERT is going to begin to try to convert locks to
            "being-used"*/
         /* Convert as many more locks as possible to "being-used".  If all
            locks have been converted, we can pass this request to
            DIRECTORY-MANAGEMENT. */

12.3.2      perform LOCK-CONVERT( TU , Req , PosInReq , Success );

         /* Execute the Req if LOCK-CONVERT was successful */
12.3.3       if Success
12.3.4          then
                    /* Send Req to DIRECTORY-MANAGEMENT for processing. */
12.3.5              perform C-SEND$MESSAGE( TU , Req );
12.3.6       end if;

12.3.7   end procedure;
```

```
12.3.2.1  procedure LOCK-CONVERT(Input  : TU , Req , PosInReq ,
                                 Output : SuccessFlag );

            /* Convert as many locks as possible for the request Req in the
               traffic unit TU.  Start at the position PosInReq in the request
               Req and convert locks until the EOR marker of that request is
               found.  If all locks are convertible to "being-used" then
               return SuccessFlag = True.  Otherwise return SuccessFlag =
               False and mark the cluster at ReqPos as "waiting"
               ( ie. Deactivate the request).  */

12.3.2.2        ReqPos  = PosInReq;
12.3.2.3        SuccessFlag = True;

12.3.2.4        while (( ReqPos <> EOR ) and ( SuccessFlag )) do

                    /* CONVERTIBLE checks to see if the lock on
                       ReqPos.ClusterId can be converted to "being-used" from
                       either "to-be-used" or "waiting".  The flag ConvertFlag
                       is set to true if the lock on the cluster ReqPos.ClusterId
                       can be converted to "being-used".  */
12.3.2.5            perform CONVERTIBLE( TU , Req , ReqPos , ConvertFlag );

12.3.2.6            if ConvertFlag /* Update the TUCT and CTUT queues  */
12.3.2.7              then      /* by setting the category to "being-used".*/
12.3.2.8                ReqPos.Category = "being-used";
12.3.2.9                perform CTUT$ADJUST-TU( TU , Req ,
                                    ReqPos.ClusterId , "being-used" );
12.3.2.10               ReqPos = ReqPos + 1;
12.3.2.11             else
                        /* Deactivate Req in TU.  This is accomplished by
                           updating the TUCT and CTUT queues by setting the
                           category to "waiting".  */
12.3.2.12               ReqPos.Category = "waiting";
12.3.2.13               perform CTUT$ADJUST-TU( TU , Req ,
                                    ReqPos.ClusterId , "waiting" );
12.3.2.14               SuccessFlag = False;
12.3.2.15         end if;
12.3.2.16       end while;
12.3.2.17 end procedure;



12.3.2.5.1  procedure CONVERTIBLE( input : TU , Req , ReqPos ,
                                   output: ConvertFlag);

            /* Given the traffic unit TU and it's request Req, can the lock on
               the cluster ReqPos.ClusterId be converted to "being-used"?

               Consider the pictorial description of the cluster queue(CTUT)
               entry for ReqPos.ClusterId given below:

               ReqPos.ClusterId : Earlier    <--   Req   -->  Later     <EOQ>
                                  Requests                    Requests

               Define the following terminology:

               Compatible Requests : Two requests are compatible if their
                   simultaneous execution will not compromise the database.
                   This occurs if their modes are both either delete, insert,
                   or retrieve.
```

Permutable Requests : Two requests R1 and R2 are permutable if the backends can execute the requests in either order, <R1><R2> or <R2><R1>, without compromising the database. The conditions which the two requests must satisfy are listed below :

i). R1.Mode = Insert and R2.Mode = Delete (or vice versa)
or R1.Mode = Insert and R2.Mode = Update (or vice versa)
or R1.Mode = Insert and R2.Mode = Retrieve (or vice versa)

and

ii). R1 and R2 are the only requests in their respective traffic units (R1 and R2 are not part of a transaction).

Convertible Lock : The request Req can convert the lock to "being-used" on the cluster ReqPos.ClusterId if the following conditions hold :

i). For each Earlier request R, Req is compatible with R or Req is permutable with R when R.Category="to-be-used".

and

ii). For each Later request R, Req is compatible with R if R.Category is "being-used".

Observations : Consider that the conditions hold for a request Req in a cluster ReqPos.ClusterId, that is the lock was converted to "being-used". There are now two cases to analyze.

1). If the lock was converted when Req was compatible with each Earlier request R. In this situation, the permutable condition for a "to-be-used" category was never utilized. Thus the effective result of granting the "being-used" lock was to permit Req to be run concurrently with other requests in the ReqPos.ClusterId.

2). If the lock was converted and there was one earlier request, R(where R.Category="to-be-used"), such that Req was permutable with R but not compatible with R, then the effective result of granting the "being-used" lock was to permit Req to be run before the request R. Thus the two requests have been permuted. Now the request R will have to wait for the request Req to finish execution before it can be run.

Lastly, the table below represents a synopsis of the definitions of Compatible(C) and Permutable(P) request pairs. An N indicates a request pair that is not Compatible and not Permutable.

	Delete	Insert	Update	Retrieve
Delete	C 1	P 3	N 2	N 2
Insert	P 3	C 1	P 3	P 3
Update	N 2	P 3	N 2	N 2
Retrieve	N 2	P 3	N 2	C 1

*/

```
            /* Initially set ConvertFlag to True.  ConvertFlag will be set to
               false only if the request Req cannot convert the lock to
               "being-used".*/

12.3.2.5.2      ConvertFlag = True;

12.3.2.5.3      ClusPos = "First position in the cluster queue
                          ReqPos.ClusterId"

            /* This while loop is used to verify condition i). of the
               definition for a convertible lock.  Looping continues while
               the flag is true and Earlier requests are being scanned.*/

            /* The while loop below implements the algorithm for a specific
               request Req.

                 for  each Earlier requests R do
                    if the R.Category is "being-used" then Req must be
                          compatible with R;
                    if the R.Category is "waiting" then Req must be compatible
                          with R;
                    if the R.Category is "to-be-used" then Req can be either
                          compatible or permutable with R;
                 end for;                                                  */


12.3.2.5.4      while ((( ClusPos.TU <> TU ) or ( ClusPos.Req <> Req ))
                                    and ( ConvertFlag )) do
12.3.2.5.5         perform  COMPATIBLE( ClusPos.Mode , ReqPos.Mode ,
                                        ConvertFlag );
12.3.2.5.6         if (( ClusPos.Category = "to-be-used" )
                              and ( not ConvertFlag ))
12.3.2.5.7            then
12.3.2.5.8              ConvertFlag = True;
12.3.2.5.9              perform  PERMUTABLE( ClusPos , TU , ReqPos.Mode ,
                                                           ConvertFlag);
12.3.2.5.10        end if;
12.3.2.5.11        ClusPos = ClusPos + 1;
12.3.2.5.12     end while;


12.3.2.5.13     if ConvertFlag  /*Condition i). in the definition of
                                  Convertible locks was satisfied. */
12.3.2.5.14       then
12.3.2.5.15         ClusPos = ClusPos + 1;  /*  Increment past the request
                                                Req in ReqPos.ClusterId. */

                  /* This while loop is used to verify condition ii). of the
                     definition for a convertible lock.  The looping con-
                     tinues while the flag is true and until all Later
                     requests have been scanned. */

                  /* The while loop below implements the algorithm for a
                     specific request Req.
                          for  all Later requests R  do
                             if R.Category is "being-used" then Req must be
                                compatible with R;
                             if R.Category is "waiting" then it doesn't
                                matter.
                             if R.Category is "to-be-used" then it doesn't
                                matter.
                          end for;                                         */
```

```
12.3.2.5.16         while (( ClusPos <> "null" ) and ( ConvertFlag )) do
12.3.2.5.17            if ClusPos.Category = "being-used"
12.3.2.5.18               then
12.3.2.5.19                 perform  COMPATIBLE( ClusPos.Mode , ReqPos.Mode ,
                                                                  ConvertFlag);
12.3.2.5.20            end if;
12.3.2.5.21            ClusPos = ClusPos + 1;
12.3.2.5.22         end while;
12.3.2.5.23     end if;
12.3.2.5.24 end procedure;
```

```
12.3.2.5.5.1 procedure  COMPATIBLE( input : Model , Mode2 ,
                                    output: CompatFlag );

             /* This procedure checks to see if the two requests represented
                by Model and Mode2 are compatible requests.  The if condition
                below actually is used to decide when two requests are not
                compatible.  The condition on the if statement represents all
                entries of the table listed in the procedure CONVERTIBLE that
                are not C's, that is all entries labelled with a 2 or a 3.  */


12.3.2.5.5.2     if  (( Model1 <> Mode2) or
                     (( Model = Mode2 ) and ( Model = "update" )))
12.3.2.5.5.3        then
12.3.2.5.5.4              CompatFlag = False;
12.3.2.5.5.5     end if;
12.3.2.5.5.6 end procedure;
```

```
12.3.2.5.9.1  procedure  PERMUTABLE( input : ClusPos , TU , ReqMode ,
                                     output: PermuteFlag );

              /*  This procedure is used to see if the two requests
                  ClusPos,Req and the request represented by ReqMode satisfy
                  the definition of permutable(see procedure CONVERTIBLE).  */

              /*  First check to see if either request is part of a Trans-
                  action */

12.3.2.5.9.2      if  (( ClusPos.TU.#REQ > 1 ) or ( TU.#REQ > 1 ))
12.3.2.5.9.3         then
12.3.2.5.9.4           PermuteFlag = False;
12.3.2.5.9.5         else
                       /* If the else clause was reached, then neither
                          request was part of a transaction.  Thus it is
                          necessary to check if the two requests are per-
                          mutable.  The condition on the if statement rep-
                          resents all entries of the table listed in the
                          procedure CONVERTIBLE that are not P, that
                          is all entries labelled with a 1 or a 2. */

12.3.2.5.9.6             if   (( ClusPos.Mode = ReqMode ) or
                                (( ClusPos.Mode <> "insert" ) and
                                 ( ReqMode <> "insert" )))
12.3.2.5.9.7                then
12.3.2.5.9.8                  PermuteFlag = False;
12.3.2.5.9.9             end if;
12.3.2.5.9.10     end if;
12.3.2.5.9.11 end procedure;
```

```
module  TUCT$;

12.2.3.1 procedure  NEW( input : TrafficUnitId ,
                         output: TU );

          /*  Add a new index element of the form :

              {  TU = the traffic unit id;
                 #REQ = the number of requests in the traffic unit; }

             The #REQ is initialized to zero.     */
12.2.3.2 end procedure;

12.2.20.1 procedure  ENTER( input : TU , TUCT-QE );

           /* Add a queue element to the TUCT table for the traffic unit TU.*/

12.2.20.2 end procedure;

12.2.22.1 procedure  EOR;

            /* Appends an end of request marker to the current traffic unit
               entry. */

12.2.22.2 end procedure;

12.2.25.1 procedure  EOT;

            /* Appends an end of traffic unit marker to the current traffic
               unit entry.*/

12.2.25.2 end procedure;

14.20.1 procedure  FREE-REQ( input : PosInReq );

          /* Remove queue elements from a traffic unit in the TUCT table
             starting at the position PosInReq and continuing until an EOR
             marker is found.  Procedure will not remove the EOR marker. */

14.20.2 end procedure;

14.3.1 procedure  FIND( input : FinishedRequestId ,
                        output: TU , Req , PosInReq );

         /* Given the FinishedRequestId find :
            1.  The traffic unit id for this request , TU.
            2.  The request number within the traffic unit, Req.
            3.  The position in TUCT of the first cluster which
                the request Req used, PosInReq.  */

14.3.2 end procedure;

14.13.1 procedure  FINDPosInReq( input : TU , Req , ClusterId ,
                                 output: PosInReq );

          /* Given a traffic unit TU, a request in TU, Req, and a ClusterId
             find the position of the queue element which contains ClusterId.
                                                                          */

14.13.2 end procedure;
```

```
14.24.1 procedure NEXT-REQUEST( input : TU ,
                                output: Req , PosInReq );

          /* Given a traffic unit TU perform the following task :

             Remove the first element in the traffic unit queue TU which is
             an EOR marker.

             Set PosInReq to the "First element in the traffic unit queue TU."

             Set Req to PosInReq.Req.

             Note : The current version of concurrency control executes
                    requests sequentially within a traffic unit.  When the
                    algorithm is modified to allow for compatibility and
                    permutability within a multiple request traffic unit,
                    the procedure to find the next "executable" request
                    in the traffic unit will be updated.                  */

14.24.2 end procedure;

end module;

module CTUT$;

12.2.19.1 procedure  APPEND( input : ClusterId , CTUT-QE );

            /*  Add a queue element to the CTUT table for the cluster
                ClusterId. */

12.2.19.2 end procedure;

14.7.1 procedure FIND-REQUEST( input : ClusterId , TU , Req ,
                               output: PosInClusQ );

         /* Returns the pointer PosInClusQ(position in cluster queue) which is
            the position in the cluster queue ClusterId where the traffic unit
            TU and it's request Req live.  */

14.7.2 end procedure;

14.8.1 procedure  FREE-QE( input : PosInClusQ );

         /*  Removes the queue element pointed at by PosInClusQ from CTUT and
             places it on a free list.  */

14.8.2 end procedure;

12.3.2.9.1 procedure  ADJUST-TU( input : TU , Req , ClusterId , CAT );

             /*  Given a request Req in a traffic unit TU and a cluster
                 ClusterId which contains the (TU , Req) pair, change the
                 category of the pair, ie (TU , Req).category to CAT.  */

12.3.2.9.2 end procedure;

end module;
```

```
module C-RECEIVE$;

8.1   procedure  MESSAGE;
        /* Receives the next message for the task CONCURRENCY and stores
           it in a buffer. */
8.2   end procedure;

9.1   procedure  SENDER( output : sender );
        /* Returns the sender name of the message in the buffer. */
9.2   end procedure;

12.2.2.1   procedure  TRAFFIC-UNIT-ID( output : TrafficUnitId );
             /*  Returns the traffic unit id. */
12.2.2.2   end procedure;

12.2.4.1   procedure  REQUEST( output : Request );
             /* Returns the request id of a request in the new traffic
                unit and all of the cluster id's needed by the request. */
12.2.4.2   end procedure;

14.2.1   procedure  FINISHED-REQUEST( output : FinishedRequestId );
           /* Returns the request id of the request that has just
              finished execution.  */
14.2.2   end procedure;
end module;

module  C-SEND$;

12.3.5.1   procedure  MESSAGE( input : TU , Req );
             /* Sends the request id of a request to the Directory
                Management process for address generation and subsequent
                execution.  */
12.3.5.2   end procedure;
end module;

module  SAVE$;

14.4.1   procedure  FINISHED-REQUEST( input : FinishedRequestId );
            /* Save the request id ( FinishedRequestId ) for possible
               later use.  */
14.4.2   end procedure;
end module;
```

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93940	2
Office of Research Administration Code 012A Naval Postgraduate School Monterey, CA 93940	1
Chairman, Code 52Hq Department of Computer Science Naval Postgraduate School Monterey, CA 93940	194
Chief of Naval Research Arlington, VA 22217	1